Regression learning on patches

Joerg Frochte
Bochum University of Applied Sciences
D 42579 Heiligenhaus, Germany
joerg.frochte@hs-bochum.de

Stephen Marsland
School of Mathematics and Statistics
Victoria University of Wellington, Wellington, NZ
stephen.marsland@vuw.ac.nz

***Abstract*—Neural networks often do poorly at representing discontinuous functions, or even just functions with rapid transitions in the response surface between closely-spaced points in feature space. However, such 'edges' in the data can be a useful way to partition the feature space in order to train specialised learners for individual regions. This is particularly beneficial where these regions are relatively simple, and hence low-complexity learners can be used successfully on them. Another benefit of such an approach is that it is easily parallelisable: the specialised learners use independent partitions of the data, and so they can be trained in parallel, while output prediction is based on the output of just one network, so there is no need to combine predictions. We introduce an algorithm to partition the data that is inspired by Finite Element Tearing and Interconnecting. Using an implementation based on a decision tree with neural networks at the leaves, we demonstrate our approach for regression learning on patches of the feature space. We use both artificial and real-world datasets to show that, in some use cases, this method can outperform conventional neural networks that see the entire feature set in the original training.**

***Index Terms*—neural networks, finite elements, regression learning, decision trees, discontinuous functions**

I. Introduction

The universal function approximation theorems of neural networks date back to the late 1980s and are well-known. They state that a single hidden layer neural network with non-linear neurons in that layer can approximate any continuous function to arbitrary accuracy [1], given a sufficiently wide hidden layer. These results have recently been extended to deep networks, where depth can replace arbitrary width [2]. However, these theorems only apply to continuous functions, and many datasets are, at best, only piecewise continuous. As well as true discontinuities, there are also places where the response function varies rapidly in the inputs, and without large amounts of training data at precisely these places, the output of a neural network can show large errors at these locations.

Instead of approximating such response surfaces using a single neural network, we investigate whether or not it can be beneficial to separate the domain into discrete patches, and train individual learners on each of these patches. Variations on such techniques have been considered in Finite Element modelling, where one seeks to approximate solutions to sets of partial differential equations (PDE) on some domain by splitting the domain into subregions and merging solutions on each of these regions. In this paper, we develop a technique inspired by a method from finite elements known as Finite Element Tearing and Interconnecting (FETI) [3] to develop a practical approach to approximating discontinuous functions, and other functions with sharp variations in the response surface, using neural networks.

There are further benefits to considering such a domain decomposition. One is that some datasets consist of disjoint regions in feature space, where the data can potentially be approximated very simply within each region (an analogy would be an image, where regions corresponding to objects such as sky or ground would be locally self-similar, but markedly different from each other). Hence a way to identify these local regions can enable relatively simple models of individual regions of the response surface, instead of building a complex monolithic model. Another benefit is that—in contrast to the finite element setting—these functions act independently on their own regions, and so can be learned in parallel, with no need to find computationally expensive ways to recombine them.

A pictorial example is shown in Figure 1, where the two-dimensional feature space can be partitioned based on discontinuities in the output variable y, so that within each patch a simple model of y can be fitted. The discontinuities between the patches make it hard for a standard neural network to learn. However, it seems intuitively clear that, if an initial stage of learning can be used to identify these regions, an independent approximation of each region would be as simple as a constant function. This approach leads to local methods that—in contrast to global approaches—will be able to approximate discontinuities between the regions more efficiently and lead to parallelised training.

Figure 2 shows a very simple example that highlights that standard neural networks do not do well at dealing with discontinuities, but also highlights an important issue. The aim is to learn a model of the function

$$y = f(x) = \text{sign}(x) + x^3, \ -1 \leq x \leq 1, \tag{1}$$

which has a discontinuity or shock in the response surface. We sampled 10,000 points without noise from this data, and then trained several different models on the data.

Both plots show the output (in red) of a single neural network with three hidden layers of sizes 16, 8, and 8 and a single linear output neuron (other neurons have sigmoidal activation). This monolithic network does a reasonable job

978-1-7281-2547-3/20/$31.00 ©2020 IEEE

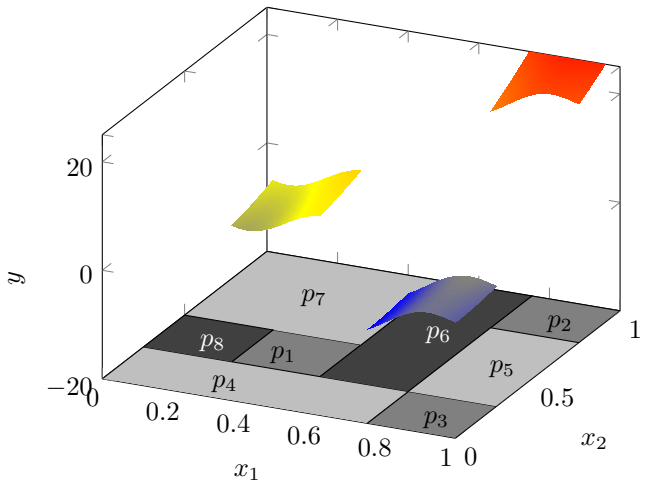

Fig. 1. Schematic of a dataset showing (very approximately) piecewise linear regions of output data y on clearly defined regions of the feature space of variables x_1 and x_2.

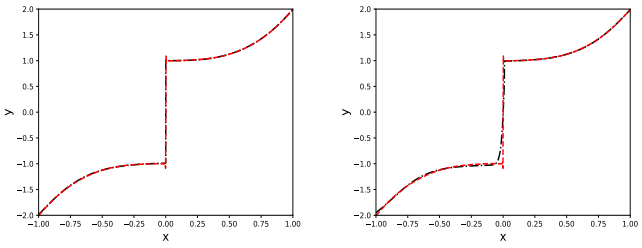

Fig. 2. Approximation with the single network (16,8,8) in red and with a model composed by two simple MLPs: *left:* with a perfect cut at $x = 0$ and *right:* with a misplaced cut at $x = 0.01$.

of approximating the function; the mean absolute error is 0.004. However, a highly complex network was required, and an effect similar to the Gibbs phenomenon (overshoot at a jump discontinuity, which corresponds to the ringing effect in images) at the discontinuity $x = 0$ is visible. The black line in the plot on the left of Figure 2 corresponds to the output of two neural networks, each with two hidden layers of two neurons each, with the networks being defined on $P_1 = [-1, 0]$ and $P_2 = (0, 1]$ respectively. This combination of two networks has just over 10% of the weights of the single network, but achieves an error of 0.003.

The main challenge of this approach is shown on the right of the figure, where the same networks are used, but with the neural networks being defined on $P_1 = [-1, 0.01]$ and $P_2 = (0.01, 1]$ instead. In this case, the model makes errors at the discontinuity, showing how important it is to identify the sharp 'edges' in the response surface accurately.

Our approach is thus to find a partition of the feature space into a set of mutually disjoint patches (which we call *data space partitioning*) and then to separate the data into the corresponding patches, and train simple, independent, models on each patch. This leads to a method that is computationally extremely cheap and inherently parallelisable. The cheapness comes from the non-linear effect of the quantity of training data on the computational complexity of neural network training. The training time of neural networks is largely driven by the complexity of the network and the number and dimension of the samples. The latter is the main aspect that makes learning on patches cheaper: each network is trained with fewer samples, and the effect is nonlinear. Of course, fewer samples also often require lower complexity networks.

Due to the fact that each of the small networks is independent and trained on separate data, all of these networks can be trained concurrently under the 'embarrassingly parallel' paradigm (in other words, it requires no programming effort or special constructions to parallelise the algorithm). The cost of the neural network training is then the maximum time to train any of the individual simple algorithms. In this paper we discuss the design requirements of such a method and introduce a simple algorithm to achieve it, which we demonstrate on both an example inspired by finite element methods and a real-world dataset concerning weather prediction.

II. Data Space Partitioning

We assume that we have a dataset $\mathcal{D} = (X, Y)$, where individual datapoints $x \in X \in \mathcal{X} \subseteq \mathbb{R}^n$ and $y \in Y \in \mathcal{Y} \subseteq \mathbb{R}$, where the response surface has spatial discontinuities or other 'edges' in the data. The aim is to *partition* the feature space $\mathcal{X}$ into subsets P_i—where by partition we mean split into mutually disjoint, non-empty subsets that cover $\mathcal{X}$—in such a way that the output variables $y \in Y$ can be approximated as simply as possible within each subset. This partitioning can be based on all the features in the data, or a subset of features can be used; such features can be chosen based on *a priori* knowledge, chosen randomly, or based on some exploration of the dataset.

The aim of our data space partitioning is to identify spatially contiguous sections of the feature space that have a relatively simple function that can explain them. One machine learning algorithm that obviously aims to perform such a split is the decision tree, which performs greedy optimisation of an information-based criterion at each stage of learning to identify the feature to split at each node of the tree. As we are interested in regression problems, CART [4] is a suitable method of training, which we use in our current implementation. These partitions can then be turned into data patches, each of which is the basis for an independent neural network trained solely on the data in that patch. We specify an upper limit k on the number of patches; this can be equal to the number of leaf nodes in the tree, or can be made smaller in order to force the algorithm to combine partitions into patches, which can be done by finding neighbouring patches that are similar.

The choice to make the partitions mutually disjoint is in contrast to the finite elements case, where overlapping subsets can be useful to increase the stability of the convergence process. For machine learning, there are two principal benefits to disjoint partitions: it simplifies the parallelism significantly, as there is no need to combine estimates, and it reduces some of

the dimensionality issues that arise otherwise. Finite element methods are generally used to model physical systems, and so are typically in two or three spatial dimensions. Suppose that we have data sampled uniformly at random in d dimensions. If we partition the data using hyper-rectangles, with a border region of p per cent of the edges lengths in each dimension, then the number of samples in the border region increases significantly as the dimensionality rises: the percentage of the data that is not in an overlapping partition is (up to corner effects):

$$\text{percentage of data in a single partition} \approx \left(\frac{100 - 2 \cdot p}{100}\right)^d.$$

For ten dimensional data with a 5% border region, only about one third of the data would be in a single partition.

We summarise our approach in Algorithm 1.

Algorithm 1 Learning on Patches (LOP)

procedure TRAIN($x \in X, y \in Y$))
Require: $k \geq 1$, $c \geq 1$ ▷ k = max # patches, c = min # samples per patch
Require: A configuration A for the local neural networks
    Divide the feature space X into patches P_i using, e.g., CART, controlled by c
    If the number of patches is $> k$, combine patches of most similarity to get k patches
    for each patch P_i **do**
        Scale/Standardise $x|_{P_i}$ and $y|_{P_i}$ $(i = 1 \dots k)$
        Train a neural network of configuration A on P_i
    end for
end procedure
procedure PREDICT($x \in X$))
    Identify the patch $P_i (i = 1 \dots k)$ that represents x
    Use the neural network associated with P_i to make a prediction
end procedure

There are a few issues with using CART, or indeed any other decision tree algorithm. Firstly, the algorithm chooses binary splits in a single feature, meaning that the partitions are axis-aligned hyperplanes, while secondly the optimisation aims to eventually replace each output with a constant function. A related observation to the second point is that, owing to the fact that the data is very unlikely to be uniformly distributed over the domain, the partioning can result in some empty or very sparsely populated patches. This means that training an ANN on these patches is not useful.

The second of these problems is relatively easy to solve by restricting the tree so that it never gets deep enough to model every leaf as a constant function. There are three parameters that could be used for this: the maximum depth of the tree, the maximum number of leaf nodes, or the minimum number of samples remaining in a leaf node. The first two are related, but can be different in the event that the tree is substantially unbalanced. We choose to use the second or third to control the amount of search. One benefit of the third is that another consideration with regard to the number of samples in each patch is that it is a factor (although there are obviously others, such as complexity of the response surface) in the speed of training. In the interests of efficiency, it then makes sense to keep the number of samples per patch as equal as possible, as we will discuss shortly.

The use of axis-aligned hyperplanes is potentially very limiting, although it is partially alleviated by the fact that we combine partitions into patches. The key issue is that any other method of performing the search will be computationally far more expensive. We experimented with a post-pruning approach that recombined patches in order to generate more flexible borders, but saw very limited improvements for significant additional computational costs; we will explore this further in future work.

We suggested that one heuristic for limiting the decision tree search is to attempt to keep the number of samples per patch as similar as possible. This leads to the question of whether or not the decision tree is necessary: an alternative, model-free, way to perform data partitioning is to specify a number of patches, and then simply choose the patches so that they have equal numbers of datapoints within them. This completes ignores the properties of the problem, but has the benefit that the ANN training is as efficient as possible, because each patch with have similar computational cost. However, the dimensionality of the feature space is then an issue. To identify appropriate patches in feature space is computationally very expensive: considering just one split in each feature means that there are 2^d possible splitting in d dimensions. A dimensionality reduction algorithm such as PCA could be used to reduce this complexity, but this only alleviates the problem rather than removing it.

Following data partitioning, all that remains is to train a neural network on each data patch. As the partitioning process has already limited the range of data in each feature, it is helpful to standardise the features for each network.

III. Experiments and Results

We present results comparing our method with a standard multi-layer Perceptron (MLP), based on the Keras/Tensorflow implementation. The tree was built using the CART regression tree "DecisionTreeRegressor" in Scikit-Learn [5]. The size of the tree can be controlled by the minimum number of samples per leaf, or maximum number of leaves, as previously discussed. The local learners for each patch were MLPs, trained using early-stopping and with a mild L_2 regulariser (1×10^{-5}). For simplicity, all these MLPs had the same size and depth; the depth matched that of the reference, monolithic, MLP, trained with the same regularisation. We report the accuracy and speed-up in training time, for a sequential implementation on a single core (Intel Core i9-9900KF Processor 3.6 GHz). As was mentioned previously, it is possible to trivially parallelise our algorithm. We therefore also give the longest training time of an individual network as a guide to the potential speed-up of simply parallelising the training of the neural networks.

A. An artificial dataset with a discontinuous feature space

Our first example is an analytical dataset that has some similarities to Figure 1. We consider a five-dimensional feature space ($x \in [0,1]^5$), with the function–particularly the discontinuities–being dominated by variables x_1 and x_2:

$$y(\mathbf{x}) = \ln(x_4+5)\left(4x_1^2+9x_2^2+\frac{x_3}{2}\right) + 2\sin(20\pi\cdot x_1x_4x_5) + P(x_1,x_2),$$

$$\text{where } P(x_1,x_2) = \begin{cases} 5, & (x_1,x_2)\in[0.35,0.6]^2 \\ 10, & (x_1,x_2)\in[0.75,1.0]^2 \\ -5, & (x_1,x_2)\in[0.75,1.0]\times[0,0.25] \\ 0, & \text{otherwise.} \end{cases} \quad (2)$$

We sampled 400,000 datapoints uniformly at random from this from this dataset, and added Gaussian noise with variance equal to 5% of the range of outputs. The data was split into 80% training and 20% testing.

Table I shows results for different settings of the number of patches (chosen by decision tree in the first set (ID 1–3), and to equalise the number of datapoints per patch in the second (ID 4–6)) and network complexity for this example, together with two monolithic neural networks (ID 7 and 8) for comparison. It can be seen that relatively complex networks were needed to achieve a reasonable mean absolute error (MAE).

Even using our sequential implementation, the LOP results are significantly faster than the monolithic networks. This is because there are fewer sample datapoints for each network, so that each epoch is cheaper. However, the networks are trained for more iterations before the learning terminates; the right of Figure 4 shows that the networks trained for over 200 epochs in all cases of LOP; in comparison the standard MLPs took 144 epochs for the larger network (ID 8 in Table I) and 200 epochs for the smaller (ID 7). The change in overall training time for these two models (132%) is almost linear with the increase in epochs (1.38), suggesting that the amount of data used for training and the number of epochs are the dominant factors. The maximal time for training a single patch shows a similar effect.

Comparing the tree-based patches with the balanced ones it can be seen that the MAE is similar in the two. This is rather surprising, since the latter does not place splits at the discontinuities. However, the maximum error is worse for the equal partitions, which presumably reflects this.

In Figure 3 we show a two-dimensional plot of the data we used in this example in variables x_1 and x_2, with colour representing the value of the target variable y, together with the residual error for different models. It can be seen that the monolithic ANN has problems with all borders of the discontinuity. In general, LOP splits along the borders, except for the upper border of the middle patch. Looking at the decomposition of the feature space into patches, it can be seen that the light green patch does not honour the border, meaning that the local model is no better than the global one at this point.

This artificial problem shows the weaknesses of both partitioning approaches. If there are hard breaks in the model then the decision tree can have trouble capturing them exactly: using the CART algorithm as a separator is no guarantee of a perfect cut. In real-life datasets that have milder changes, the precise position of the cut seems to matter less, as we shall see.

B. Heat distribution data

In this example we consider the kind of problem for which finite elements are commonly used. We simulate the distribute of heat as it is conducted from a heat source located at $(h_x, 0.1)$ with a diameter of 0.025 cm for a time $20\text{ s} \le t_h \le 120\text{ s}$. The material is a two-dimensional copper square with an edge length of 0.1 cm. The letters "AI" have been removed from the centre of the square; the resulting air gap transports the heat more slowly than the copper. The heat distribution changes at the boundaries of the letters, but it is continuous.

The goal is to predict the temperature at a point x, y in the square for given h_x and t_h. Therefore, we have four-dimensional feature space (x, y, h_x, t_h). We created $1,929,008$ data records for training and $492,707$ for testing. The testing set does not share any records with the same parameter tuple (h_x, t_h), so that the problem is not a simple spatial interpolation.

As a sample configuration, we created a set of patches by using the two-dimensional (x, y) coordinates as inputs to CART, and specified that the tree retained at least 50,000 training samples per leaf node. This resulted in a tree with 31 leaf nodes (depth 5). Data corresponding to each of these 31 patches was then used to train a separate neural network with three hidden layers of 25 neurons each. As comparator we used a single neural network with three hidden layers of 100 neurons each. Figure 5 shows that both approaches have trouble at the borders of the letters, but the errors are more noticeable for the global ANN. This is borne out by the MAE, which is 0.017 for the LOP approach, and 0.031 for the global ANN. Further, sequential training took 130 minutes for LOP, as opposed to 270 for the single ANN. This is despite the fact that in this case the LOP set-up has more than double the total number of parameters: 44,950 as opposed to 20,800. The most expensive patch took around 400 seconds (just under 7 minutes) to train, meaning that the parallel implementation would be significantly faster again.

C. A weather dataset with a continuous model illustrating the feature selection for patch building

As a final example of our approach, we used some real-world data: prediction of the hourly air temperature based on a set of atmospheric data. The data came from the Murdoch University Weather Station (http://wwwmet.murdoch.edu.au/downloads). We used the data from 2005 to 2009 as training and validation data (a total of 196,055 records), each comprising twelve input features (month, day, wind speed and direction (10 minute average and standard deviation),

ID	# patches	Network Architecture	MAE	Max error	Total time (min, %)	Max single time (min, %)
1	20	(75,75) tree	0.42	7.77	9.5 (49%)	2.3 (12%)
2	26	(50,50) tree	0.51	8.19	9.8 (50%)	2.1 (11%)
3	40	(30,30) tree	0.62	8.91	9.6 (49%)	1.2 (6%)
4	16	(75,75) equal	0.41	8.9	10.8 (55%)	1.4 (7%)
5	25	(50,50) equal	0.54	8.9	9.7 (50%)	0.9 (5%)
6	36	(30,30) equal	0.63	8.4	9.5 (49%)	0.6 (3%)
7	1	(75,75) mono	0.48	9.6	19.4 (99%)	19.4 (99%)
8	1	(150,150) mono	0.39	10.1	19.5 (100%)	19.5 (100%)

TABLE I

RESULTS FOR DIFFERENT NETWORK CONFIGURATIONS SOLVING THE ANALYTICAL DATASET EXAMPLE PROBLEM

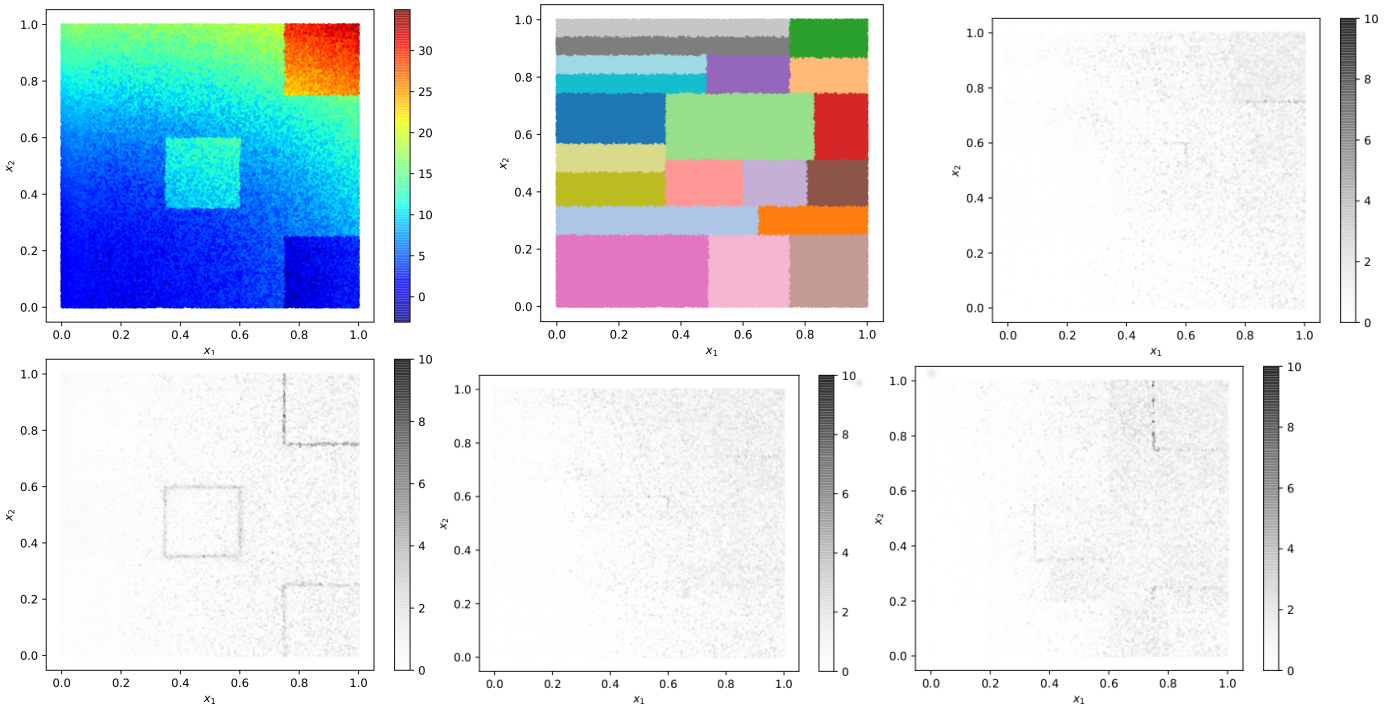

Fig. 3. *Upper left:* Plot of the $x_1 - x_2$ plane of the analytical dataset with colour representing the value of the target variable y. *Upper middle:* set of 20 patches chosen by the LOP algorithm (ID 1). *Upper right:* Plot of the residual error using tree-partitioned LOP (ID 1). *Lower left:* Plot of the residual error using equal-point LOP (ID 5). *Lower middle:* Plot of the residual error using tree-partitioned LOP (ID 2). *Lower right:* Plot of the residual error for a monolithic MLP (ID 7).

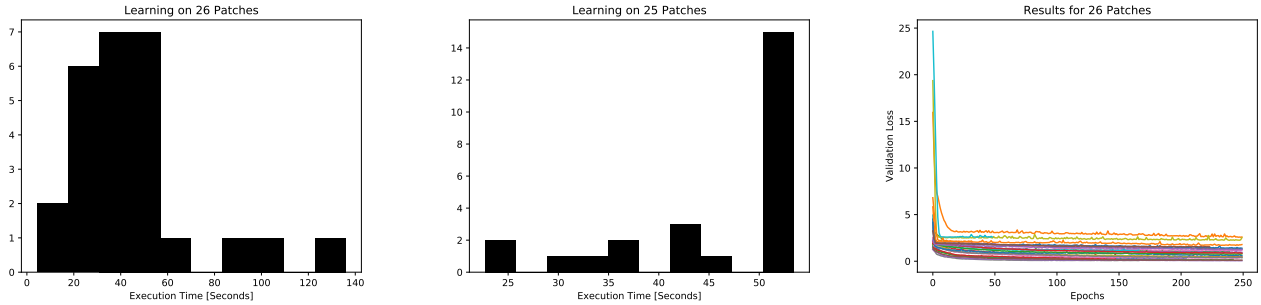

Fig. 4. For the analytical dataset, execution time for LOP learning with *left:* tree-based partitioning with 26 patches (ID 2 in Table I) and *middle:* equal dataset-based partitioning (ID 5). Both have an ANN with two hidden layers of 50 nodes each for each patch. *Right:* validation loss for ID 2

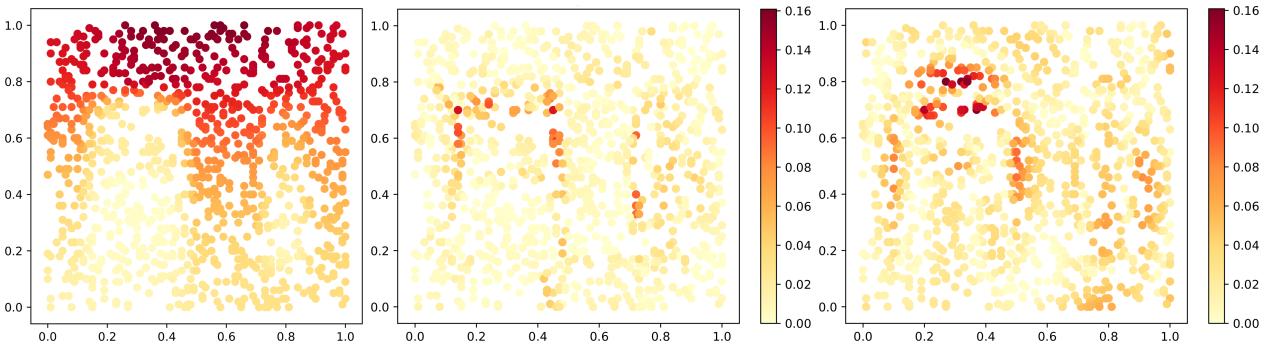

Fig. 5. *Left:* A sample normalised heat distribution. *Middle:* Error plots for normalised target values at $t_h = 71$ secs at position 0.0515 m for LOP with 31 patches. *Right:* Error for monolithic ANN with architecture (4,100,100,100,1).

# patches	Network Architecture	Feature Separation	MAE	Max Error	Total time (min, %)	Max single time (min, %)
25	(13,) tree	4 of 4 (PCA)	2.59	16.09	8.0 (55%)	0.9 (6%)
25	(13,) tree	6 of 12	1.95	9.21	6.3 (43%)	0.6 (4%)
25	(13,) tree	12 of 12	1.91	11.35	6.7 (46%)	0.6 (4%)
24 (3,2,2,2)	(13,) equal	4 of 4 (PCA)	2.62	18.72	7.3 (50%)	0.7 (5%)
36 (3,3,2,2)	(13,) equal	4 of 4 (PCA)	2.65	17.04	8.0 (46%)	0.5 (3%)
32 of 64	(13,) equal	first 6 of 12	2.01	35.03	7.7 (55%)	0.8 (6%)
28 of 64	(13,) equal	selected 6 of 12	2.07	23.88	8.2 (57%)	1.0 (7%)
22	(500,10) mono	-	1.85	28.7	14.5 (100%)	14.5 (100%)

TABLE II
RESULTS FOR DIFFERENT NETWORK CONFIGURATIONS LEARNING THE WEATHER DATASET

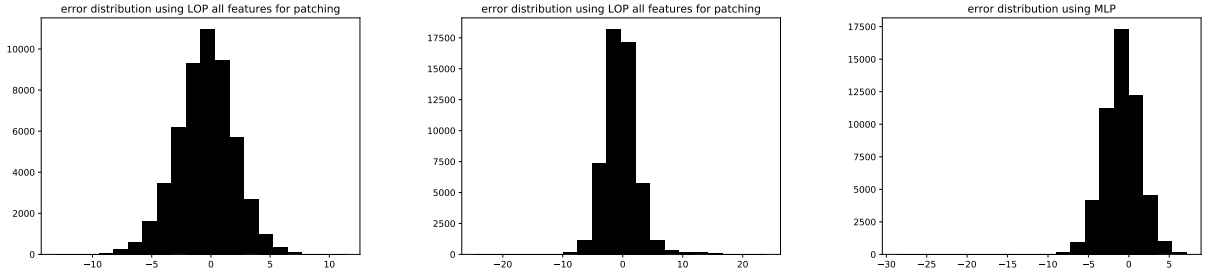

Fig. 6. Plots of the error distribution for (*left* to *right*): the LOP tree with all 12 features, the equal dataset partitioning on all 12 features, and the monolithic MLP. Note that the axis scalings differ between them, as the LOP tree produces a significantly lower error rate.

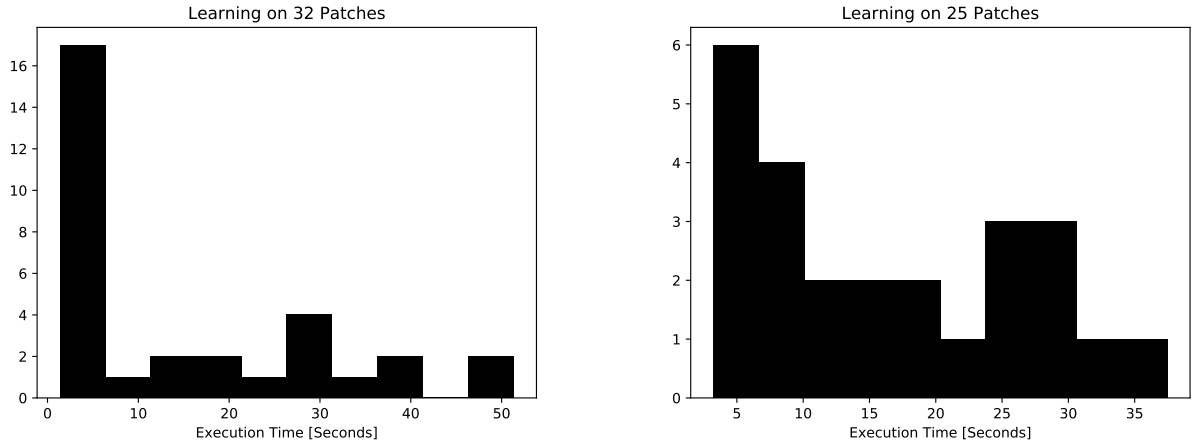

Fig. 7. Plots of the execution time for LOP with (*left*:) equal dataset partitioning, and (*right*:) tree-based partitioning. Again, note that the y-axis differs between the plots. While there are some very expensive patches for the equal partioning, most are very quick to train.

evaporation, barometric pressure (3 measures), rainfall to date and relative humidity) and a single output (temperature). The test data was taken from 2010 (51,857 records).

To build the patches, we selected six of the twelve features (month, hour, wind speed, wind direction average and standard deviation, rainfall) and built a tree with a minimum number of $6,000$ samples per leaf node, leading to 25 leaf nodes, each of which was used as a patch. In this test case the networks on each leaf node are very small: just one hidden-layer, with 13 neurons. Using this set of networks on the 25 patches leads to a MAE of 0.0399. The monolithic network had two hidden-layers: the first with 500, and the second with 10 neurons. The result is a MAE of 0.0510. It took just under 20 minutes to train the dense network, and just under 10 minutes to train all of the small networks sequentially, with the longest taking 38 seconds.

We also used LOP with all 12 features. The patches that were found this way were very similar, and the MAE of the final method was not significantly different (0.0389).

In low dimensional spaces (two or three dimensions) the choice to perform equal distance patching can perform quite well. But in higher dimensions it is not feasible to split each feature axis even once. In this problem which has 12 input dimensions, we would end up with $2^{12} = 4096$ patches. Instead, we consider a subset of the features; considering just the first 6 means that there are 64 patches. However, most of them are empty, since partitioning the data in this way highlights how much of input feature space contains no data points. In this example, just 33 patches have data and so can support their own ANN. This suggests that the higher the dimensionality of the input space, the more a tree is a better way to partition it.

It is also possible to use PCA to use PCA to reduce the feature space before learning. By reducing the feature space in this way it is far less likely that there will be empty patches. For the equal patch method there were two different PCA configuations, with the difference being the number of split points allowed for each feature. However, it can be seen that the MAE is less good with this method.

IV. Relevant Literature

The basis of our method is a way to partition the feature space so that individual patches can be approximated by relatively simple learners. In this implementation described in this paper we mainly use a decision tree to perform this partition, although as we discuss, other methods could be used. However, this makes our approach most related to model trees [6], which build single decision trees and then prune them back by considering the replacement of each leaf with a linear regressor, progressing back up the tree.

There have been other relevant extensions of decision trees, such as putting linear regressors on the leaves of a tree [7], creating a look-ahead linear regression tree [8], or seeking to compute a globally convex regression tree by partitioning a dataset by devising a tree that has linear outputs at the leaves [9]. However, these methods do not consider the general approach that we describe here, because they focus on enhancing the tree as a learning algorithm, while our approach uses it mainly for the partition of the feature space.

Another set of related approaches are the soft decision tree [10], [11] and its variants such as the adaptive neural tree [12]. In contrast to our aim of partitioning the feature space prior to most of the learning, these approaches effectively turn the tree itself into a neural network, so that all the leaves have performed computation on the features close to the root of the tree, and have less in common at lower levels of the tree. Conceptually these models can be traced back to the Hierarchical Mixture of Experts model [13].

In contrast to ensemble methods, such as boosting [14] and random forests [15] (and see e.g., [16] for an overview), our method does not aggregate the output of many learners, but produces specialised learners that are only trained on data from specific subsets of the feature space. Hence prediction consists of identifying the relevant patch of feature space, and then using only the local model associated with that patch. This makes the process more efficient in both training and testing.

One motivation for our approach was the methods of domain decomposition used in the numerical solution of PDEs; see e.g., [3]. For parallel training of dense neural networks, there are two main approaches: data-parallelism (mainly batch-splitting) and model-parallelism. Batch splitting, see e.g., [17], suffers from several problems when training very large models. On the other hand, the analysis of the graph of deep neural networks mostly used for model-parallelism rely on the structure of the neural network to find paths that can be separated, which may not exist for dense networks, with their highly connected structure. In terms of data-parallelism and model-parallelism our approach combines both and is therefore able to lead to a much higher potential. We divide the data into patches and build a specific model on each patch. The main difference to standard model-parallelism is that we do not parallelise a model for the full feature space but build several single models. Beyond this, standard approaches like batch-splitting are still an option for each of the specialised models.

V. Conclusion

We have presented a method of learning by partitioning of the feature space that is computationally cheap and inherently parallelisable, since it works by training completely independent models on partitions of the data, something that can be trivially parallelised for substantial savings in elapsed training time. In our implementation here the method is based on standard machine learning algorithms. On the datasets we have tested it on it produces results that are at least comparable with far more computationally complex models.

There are still several questions that we have not yet considered fully. One of the most important is to find other computationally cheap ways to find partitions; as discussed earlier, CART is effective, but only works for axis-aligned cuts. It might be that a method from change detection, or a simple clustering in feature space could be useful, and we

will test this in the future. Another interesting question is concerned with model selection. We have used the same neural network architecture for all partitions in this implementation, but heuristics and methods from AutoML could be used to choose different models for each of the local partitions. It would also be possible to let the networks grow adaptively on individual patches if the results were not sufficiently accurate on them. In this way, the complexity of the learner on each patch could be specialised.

In order to aid comparisons with the monolithic ANN, in this paper we used a sequential implementation. However, we have highlighted the benefits of our algorithm for parallel training. Our results demonstrate that by learning on patches one can achieve the same accuracy, and often even better, than with monolithic neural networks. However, the embarrassingly parallel nature of the approach means that the speedup that is possible using this approach is substantial even without any programming effort.

The method is most effective where there is a clear edge between regions of the feature space, although in these cases it is particularly important that the position of the edge is identified accurately. It may be that this has particular benefits for classification, rather than regression, problems, and we will investigate this soon. We will also consider ways to detect when new data is presented in currently sparsely occupied regions of the feature space.

ACKNOWLEDGMENT

SM is supported by Te Pūnaha Matatini, a New Zealand Centre of Research Excellence in Complex Systems.

REFERENCES

[1] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals, and Systems (MCSS)*, vol. 2, no. 4, pp. 303–314, 1989.

[2] Z. Lu, H. Pu, G. Wang, Z. Hu, and L. Wang, "The expressive power of neural networks: A view from the width," in *Advances in Neural Information Processing Systems*, 2017.

[3] C. Farhat and F.-X. Roux, "A method of finite element tearing and interconnecting and its parallel solution algorithm," *International Journal for Numerical Methods in Engineering*, vol. 32, no. 6, pp. 1205–1227, 1991.

[4] L. Breiman, J. H. Friedman, and C. J. Olshen, R. A.and Stone, *Classification and regression trees*. Wadsworth & Brooks, 1984.

[5] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[6] E. Frank, Y. Wang, S. Inglis, G. Holmes, and I. H. Witten, "Using model trees for classification," *Machine learning*, vol. 32, no. 1, pp. 63–76, 1998.

[7] A. Dobra and J. Gehrke, "SECRET: A scalable linear regression tree algorithm," in *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2001.

[8] D. S. Vogel, O. Asparouhov, and T. Scheffer, "Scalable look-ahead linear regression trees," in *Proceedings of KDD*, 2007.

[9] L. A. Hannah and D. B. Dunson, "Multivariate convex regression with adaptive partitioning," *Journal of Machine Learning Research*, vol. 14, pp. 3261–3294, 2013.

[10] A. Suárez and J. F. Lutsko, "Globally optimal fuzzy decision trees for classification and regression," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 21, no. 12, pp. 1297–1311, 1999.

[11] N. Frosst and G. E. Hinton, "Distilling a neural network into a soft decision tree," in *CEX workshop at AI*IA*, 2017.

[12] R. Tanno, K. Arulkumaran, D. Alexander, A. Criminisi, and A. Nori, "Adaptive neural trees," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 97, 2019, pp. 6166–6175.

[13] M. I. Jordan and R. A. Jacobs, "Hierarchical mixtures of experts and the EM algorithm," *Neural Computation*, vol. 6, no. 2, pp. 181–214, 1994.

[14] R. Schapire, "The boosting approach to machine learning: An overview," in *Nonlinear Estimation and Classification*, D. D. Denison, M. H. Hansen, C. Holmes, B. Mallick, and B. Yu, Eds. Berlin, Germany: Springer, 2003.

[15] T. K. Ho, "Random decision forests," in *Proceedings of the 3rd International Conference on Document Analysis and Recognition*, 1995, pp. 278–282.

[16] Z.-H. Zhou, *Ensemble Methods: Foundations and Algorithms*. Chapman and Hall/CRC, 2012.

[17] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young *et al.*, "Mesh-tensorflow: Deep learning for supercomputers," in *Advances in Neural Information Processing Systems*, 2018, pp. 10 414–10 423.