University of Exeter

Can statistics help us to understand deep learning?

Johannes Smit

May 2019

Abstract

Machine learning and deep learning have expanded very quickly and seen many successes. Due to their hierarchical structure, deep neural networks are very difficult for humans to understand, which could be a problem as machine learning algorithms take greater control of our lives. This project uses a simple example of a deep neural network learning a nonlinear function to test various statistical methods including multiple regression and Gaussian process regression.

Acknowledgements

Contents

Chapter 1

Introduction

Since the turn of the millennium, machine learning, and particularly deep learning have been able to solve many problems that were once thought impossible for a computer and surpass traditional algorithms in many contexts. Machines can now identify objects (Li, Katz and Culler 2018), beat humans at PSPACE-hard games such as Go (Chao *et al.* 2018) and even drive cars (Gerla *et al.* 2014). These tasks are so complex that human designed algorithms quickly become infeasible, so instead, we design a process that uses datasets of correct examples to teach the machine how to respond to patterns (Murphy 2012, p. 1). One of these machine learning algorithms is the artificial neural network, which takes inspiration from the neural structure of the biological brain. Neural networks consist of units called 'neurons', which individually perform a simple nonlinear operation, but when interconnected can understand more complex structures (LeCun, Bengio and Hinton 2015, p. 436). Due to the abstracted nature of a machine learning algorithm's calculations, it can be difficult to provide a human understandable explanation for its reasoning, and often it is impossible.

As these algorithms are placed in more real world scenarios and in control of important infrastructure and even human lives, the need to understand what is happening inside the 'black box' becomes more important. Autonomous vehicles are becoming more common, and will face corner cases which no human could have predicted and preprogrammed a response. In cases where a driverless car does something unforeseen, it will be useful to have some understanding of what the algorithm was 'thinking' when it made

the decision. machine learning systems which aid in probation and parole decisions in the U.S. are now being trialled for sentencing, but have come under a lot of criticism for being racially biased (Christin, Rosenblat and Boyd 2015). The question of how an algorithm can be 'held accountable' has been raised (Christin, Rosenblat and Boyd 2015, p. 9), but it is not known what accountability looks like for an artificial mind.

In this project, I will use regression techniques including multiple regression with Fourier basis functions and Gaussian process regressions to model the output of a deep artificial neural network and try to recover the.

To-do: *finish that paragraph*

Chapter 2

Machine learning

The sudden rise in success of machine learning and deep learning is down to an increase in computing power, particularly graphics processing units (GPUs) which are designed to perform matrix operations very efficiently. This has allowed for more complex models to be trained in reasonable times, particularly hierarchical models.

2.1 Artificial neural networks

One of the most popular types of machine learning algorithm is the artificial neural network, a very general algorithm that has seen great sucess in performing regression and classification tasks, as well as in other contexts such as reinforcement learning. It was chosen for this project because of its popularity and widespread and varied use cases.

2.1.1 The perceptron

In 1957, psychologist Frank Rosenblatt proposed a stochastic electronic brain model, which he called a 'perceptron' (Rosenblatt 1957). At the time, most models of the brain were deterministic algorithms which could recreate a single neural phenomenon such as memorisation or object recognition. Rosenblatt's biggest criticism of these models was that while a deterministic algorithm could perform a single task perfectly; unlike a biological brain, it could not be generalised to perform more tasks without substantial reprogramming. He described deterministic models of the brain as 'amount[ing]

simply to logical contrivances for performing particular algorithms [...] in response to sequences of stimuli' (Rosenblatt 1958, p. 387). Another way he wanted his synthetic brain to mirror biological brains was the property of redundancy, the ability to have pieces removed and still function, which is not possible for deterministic algorithms where even a small change to a circuit or a line of code can stop all functionality.

It was a commonly held belief that deterministic algorithms 'would require only a refinement or modification of existing principles' (Rosenblatt 1958, p. 387), but Rosenblatt questioned this idea, believing that the problem needed a more fundamental re-evaluation. The crucial idea was the 'perceptron', which could – through repeated training and testing – receive a set of inputs and reliably give a correct binary response. The perceptron was later generalised to the concept of the artificial neuron (also called a 'unit'), which, instead of only giving a binary output, maps a number of real inputs to a single real output value.

2.1.2 Artificial neurons

A neuron is defined by its weight vector $\boldsymbol{w}$, its bias b and its activation function (or 'limiter function') ϕ.

The stages of a neuron with n inputs, seen in Figure 2.1, are:

1. For an input vector $\boldsymbol{x} \in \mathbb{R}^n$ and a weight vector $\boldsymbol{w} \in \mathbb{R}^n$, take a weighted sum of the inputs, called a 'linear combiner'.

$$\text{linear combiner} = \sum_{i=1}^{n} x_i w_i$$

2. Then, the bias $b \in \mathbb{R}$ is added, which translates the output to a suitable range. The bias controls how large the weighted sum needs to be to 'activate' the neuron, so this is called the pre-activation (v).

$$\text{pre-activation} = v = \sum_{i=1}^{n} x_i w_i + b$$

3. Finally, the activation function ϕ is applied, which restricts the output to a range and introduces nonlinearity to the system. The activation

function must be differentiable if the gradient descent method is used (see Section 2.1.4).

$$\text{neuron output} = \hat{y} = \phi\left(\sum_{i=1}^{n} x_i w_i + b\right)$$

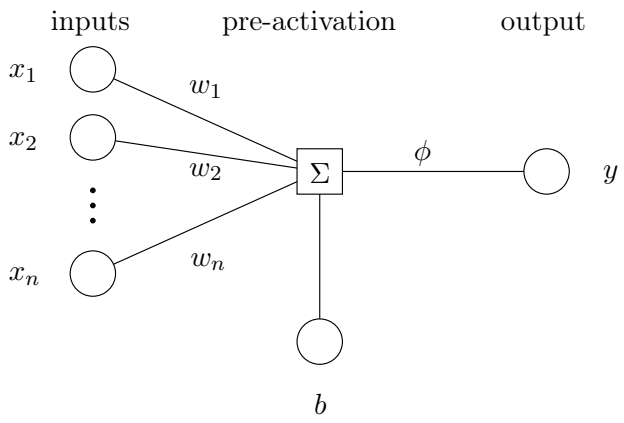

Figure 2.1: A diagram of an example artificial neuron.

2.1.3 Activation functions

Early examples of artificial neurons were built in hardware with the output being a light that was either on or off. This is equivalent to the sign function, which is now only used for binary classification problems. It is not useful for connecting to another neuron, as information about the magnitude of the output is lost. Commonly, the sigmoid function $S(x) = 1/(1 + e^{-x})$, the tanh function or the 'softmax' function (a generalisation of the logistic function to higher dimensions) are used. More recently, a popular activation function for deep learning (see Section 2.2) is the rectified linear unit (ReLU) function (Ramachandran, Zoph and Le 2017), which is defined as the positive part of its input, i.e., $\text{ReLU}(x) = \max(0, x)$. The ReLU function was designed to be analogous to how a biological neuron can be either inactive or active, although why it works as well as it does is not well understood. If the activation function is the identity function, then optimising a neuron is equivalent to performing linear regression.

2.1.4 Training

In the case of a 'feedforward' neural netowrk, the neurons are connected in sequential groups called layers, where each layer only receives information from the previous layer and only passes information to the subsequent layer. Convolutional and recurrent neural networks also exist in which the layers are not connected in series. A fully connected (or dense) neural network is one where every neuron in a layer is connected to every neuron on the subsequent layer, as seen in Figure 2.2.

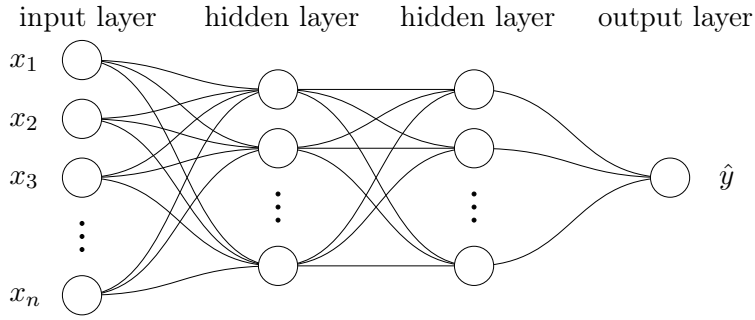

Figure 2.2: An example of the structure of a fully connected feedforward neural network.

An efficient method to optimise (or 'train') the weights in a neural network was not known until Linnainmaa (1970) published his method of automatic differentiation, which was first applied to neural networks in 1982 by Werbos (1982). The weights $\boldsymbol{w} = w_1, \ldots, w_N$ are randomly initialised. One step (called an 'epoch') in the backpropagation algorithm is defined as:

1. Input a dataset of n datapoints $X = \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ with corresponding targets $y_1, \ldots, y_n$.

2. Repeat instructions 3–7 for each datapoint $i = 1, \ldots, n$.

3. Use the training datapoint $\boldsymbol{x}_i$ to make a prediction $\hat{y}_i$.

4. Calculate the squared error for this datapoint ε_i by comparing the prediction $\hat{y}_i$ to the target y_i: $\varepsilon_i = (\hat{y}_i - y_i)^2/2$.

5. To reduce this error, the weights must be altered, as the target is fixed and the prediction cannot be directly changed. Calculate the gradient

of the error $\Delta_j = \partial\varepsilon_i/\partial w_j$ with respect to each of the weights $j = N, \ldots, 1$. The error at any particular layer only depends on the output of the subsequent layer and the weights connecting those two layers. By iteratively applying the chain rule, the calculation can propagate backwards though the layers.

6. The vector $\boldsymbol{\Delta} = (\Delta_1, \ldots, \Delta_N)$ represents the direction in N-dimensional weight-space that will produce the steepest increase in the error ε_i, so a step in the direction $-\boldsymbol{\Delta}$ will most steeply reduce the error.

7. Update the weights $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta\boldsymbol{\Delta}$, where η is the step-size hyperparameter, which controls how quickly the algorithm converges.

The backpropagation algorithm will adjust the weights until either the error reaches below a certain threshold or the maximum number of epochs is reached.

This type of method is called 'stochastic gradient descent' (also called 'online gradient descent' or 'iterative gradient descent'). Each datapoint moves the weights in a slightly different direction, so the optimisation process zigzags towards the optimum, rather than taking the steepest path. An alternative is 'batch gradient descent', where the sum of the squared errors for all the training data is calculated:

$$\boldsymbol{\varepsilon} = \frac{1}{2}\sum_{i=1}^{n}\left(\hat{y}_i - y_i\right)^2.$$

This means each epoch consists of only one very efficient step, and so will converge in fewer epochs, however each epoch takes so long that this takes significantly longer to finish training. In practice, 'minibatches' are used, where a subsample of fixed size of the datapoints are used to train the model. This allows for a balance of training time and model quality, as larger batches take longer to train and smaller batches are more susceptible to noise (Thoma 2017, p. 59).

Estimating the parameters of a neural network using any type of gradient descent is guaranteed to find a local optimum given enough time, but is not guaranteed to converge to a global optimum. It is quite rare for the algorithm to get trapped in a local minimum, but a more common problem is getting

stuck at saddle points where the gradient is zero (LeCun, Bengio and Hinton 2015, p. 438).

2.2 Deep learning

Although wider neural networks with many neurons on each layer had limited success in the 20th century, increasing the number of layers resulted in a neural network that was impractical to train using techniques and hardware of the time. It was only in the 1990s that training these deep neural networks became feasible, and then saw widespread success in the 21st century (Schmidhuber 2015, p. 86). Deep neural networks 'can implement extremely intricate functions of its inputs that are simultaneously sensitive to minute details' (LeCun, Bengio and Hinton 2015, p. 438).

Making a neural network deeper is not always a good idea. As neural networks get deeper, they reach the 'vanishing gradient problem', where the gradient at each layer gets smaller as the backpropagation algorithm gets closer to the input layers until the changes are insignificant. Additionally, a neural network with too many parameters for the size of the training dataset can simply 'remember' the training data, i.e., overfit and reproduce the data, which does not find any meaningful patterns.

Deep learning has even made its way into consumer products. Many modern phones have facial recognition software built in, and artificial intelligence (AI) voice assistants are becoming more common in homes. Deep learning is also becoming more accessible, for example with the release of Google's powerful 'Tensorflow' engine for deep learning (Abadi *et al.* 2016), which is now available as 'Keras', a user friendly package for Python (Chollet *et al.* 2015) and R (Allaire and Chollet 2018).

2.3 Example

To demonstrate the possibility of using statistical methods to understand the process of deep learning, consider use a simple nonlinear function $f(x) = x + 5\sin(x) + \epsilon$, where ϵ is iid Gaussian noise $\epsilon \sim \mathcal{N}(0, 0.1)$. This simple function was chosen so that a deep neaural net would easily fit it well, and then any attempts at understanding the neural network can be easily compared to the

'true' function. 256 evenly spread datapoints were drawn from this function, seen in Figure 2.3.

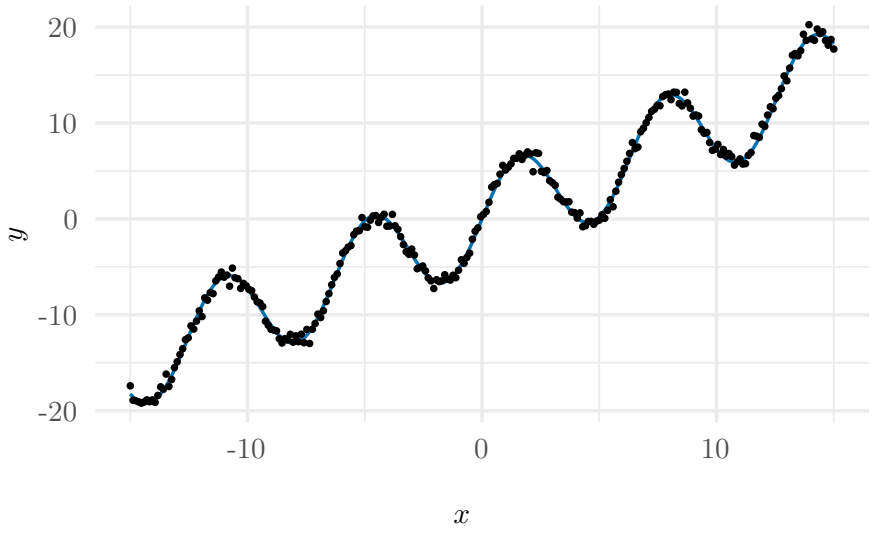

Figure 2.3: The true function $f(x) = x + 5\sin(x)$ (blue) and the 256 training points (black).

2.3.1 Training a neural network

This function was learnt using the `keras` package for R to create and train a neural network.

The number of hidden layers in the neural network was increased from 0 (linear regression) until the fit seemed reasonable at 8 hidden layers with 10 neurons each. The number of neurons in each layer was kept relatively small, as this particular problem is likely to require more levels of abstraction (depth) rather than an ability to store lots of information. The tanh activation function was at first chosen semi-arbitrarily due to its trigonometric-ness/trigonometric association/trigonometric properties?, but after being compared to the ReLU, sigmoid and softmax activation functions, it was determined to be the best (see Section 2.1.3 for a description of each of these functions). A linear activation function was required on the final output neuron so that the neural network's predictions were not limited to the range of the activation function.

The final architecture of the neural network was a fully-connected, feed-

forward neural network with 8 hidden layers, each with 10 neurons and the tanh activation function, and an output layer with a linear activation function. The neural network was trained using both batch gradient descent and minibatches of 32 datapoints. The choice of gradient descent algorithm did not make much difference to the fit, likely due to the simplicity of the function being learnt, although batch gradient descent was selected as it performed slightly better. The neural net was trained for 10,000 epochs to grantee that the weights had stably converged and the training had finished. The input data was split into 80% training data and 20% validation data.

To-do: *expand on split*

The result of this learning is seen in Figure 2.4.

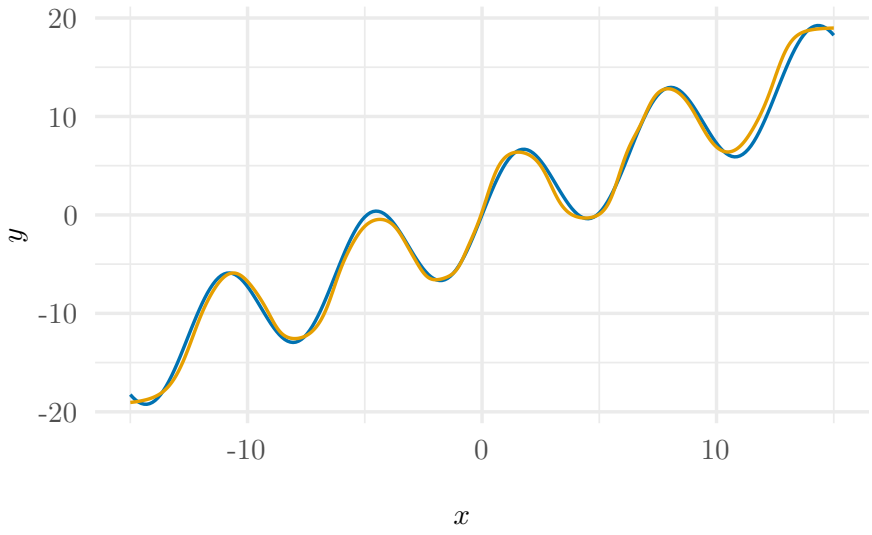

Figure 2.4: The true values (blue) and the neural network's predicted values (yellow).

2.3.2 Ordering of the datapoints

Due to the gradient descent used to train the model, if the order of the datapoints is not randomised, then the optimisation algorithm is significantly more likely to get stuck in a local minimum or saddle point where the gradient is zero. This can be seen in step 2 in Section 2.1.4, where the datapoints are

iterated over sequentially. An example of this is seen in Figure 2.5, where the same neural network has been trained on the example data which has been ordered in three different ways: increasing x, decreasing x and randomised. It seems that the first datapoints have the largest effect on the training of the model, leading to the weights reaching a local minimum where

> **To-do:** *finish this section and add the fourth sort*

Bengio *et al.* (2009) developed a technique called 'curriculum learning' which uses this fact to improve training by starting with easier training examples and slowly working towards more difficult training examples. In a similar way to how neural networks were inspired by biological phenomena, this was inspired by how humans are taught simple examples before moving on to harder tasks.

For example, when trained on a text dataset, the algorithm was trained on only texts using the vocabulary of 5,000 most common words. The vocabulary was expanded by 5,000 more words at intervals, allowing the algorithm to learn new words once it had mastered simpler words. When compared with an algorithm that had no curriculum, the curriculum-trained model took longer to reach the same error rate as it spent time early on focusing only on simple examples, but achieved a better error rate once the vocabulary was fully expanded.

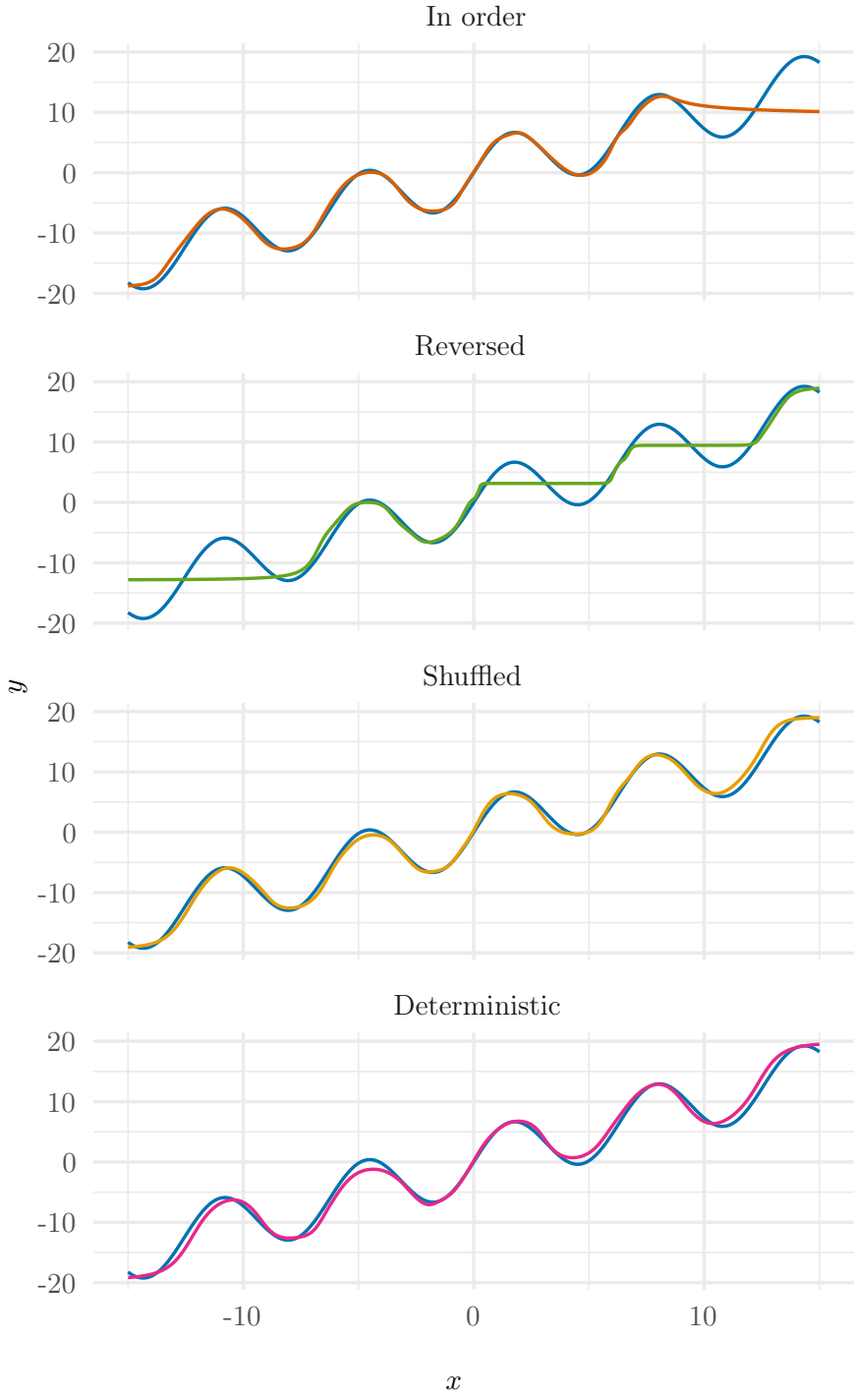

Figure 2.5: The output of the same neural network trained on the same data but ordered differently: in order (orange), reversed (green) and shuffled (yellow) compared to the true function (blue).

Chapter 3

Opening the black box

Neural networks take a long time to train (calculus is hard) but can make predictions very quickly (arithmetic is easy). This means that once trained, we can very quickly get lots of data to give to the regression model or GP or whatever. This means we're not limited by a lack of data from the neural network.

> **To-do:** *rewrite that whole paragraph*

3.1 State of the art

The fact that predicting from a neural network consists of many simple operations in a hierarchical structure suggests that it could be possible to mathematically or statistically explain or approximate their output.

When neural networks were originally trained on image classification problems, it was thought that each layer would successively combine features from the previous layer to form more complex structures. For example, a common machine learning problem is designing a neural network for classifying handwritten digits where the input is a vector of length n^2 representing the brightness of each pixel in an $n \times n$ image. A designer might plan for the first layer to pick out clusters of pixels, then the next layer would combine adjacent clusters into lines, the next layer would combine lines into corners, and the final layer would identify combinations of lines and corners as digits. However, once a neural network is trained on such a problem, the result is

always that each neuron is activated by a seemingly random set of pixels, yet the neural network seems to perform well.

> **To-do:** *finish this paragraph*

Mahendran and Vedaldi (2014) trained a deep neural network on images and attempted to 'invert' it to reconstruct the original inputs. They found that each successive layer abstracts the data, but isolating the specific neurons responsible for specific objects or ideas was more difficult.

Sensitivity analysis has also been used to evaluate the importance of each input to a neural network in the final output, and the results visualised as a decision tree where each branch represents a range of input values (Cortez and Embrechts 2013, p. 13).

There appears to be no literature on using Gaussian processes (GPs) as a way to better understand how deep learning algorithms work or as a way to understand their structure, although there has been a fierce debate over the advantages of both neural networks and GPs since at least the 1990s. Rasmussen (1997, pp. 65–66) gave evidence that GPs are a valid replacement for neural networks in nonlinear problems with fewer than 1000 datapoints, although due to advances in processing power, neural network algorithms and GP techniques, this recommendation will likely have changed. MacKay (1997, p. 25) gave an example of GPs being used for binary classification, in a similar way to neural networks. When Gaussian process regression (GPR) and neural networks are compared, the main advantages mentioned are usually faster convergence and confidence or credible intervals for the predictions, which neural networks cannot do (Herbrich, Lawrence and Seeger 2003).

3.2 Multiple regression

One method of opening the black box of machine learning is to use multiple regression on a series of basis functions. Many basis functions are calculated and traditional linear regression is used on the resulting matrix of predictors in an attempt to uncover which are the 'true' functions. This method requires that enough basis functions are used that the correct basis functions are in

the model. The number of basis functions can be increased arbitrarily, but this will increase the fitting time. This technique also requires regularisation to reduce the number of terms in the model, as there will likely be many terms with small coefficients that are determined to be significant.

> **To-do:** *Danny seemed unconvinced by this. Find a source to legitimise this.*
> *Why does this not violate the assumption that our predictors are independent?*

In this case, the basis functions x, x^2, $\sin(x)$, $\sin(2x)$, $\sin(x/2)$, $\cos(x)$, $\cos(2x)$ and $\cos(x/2)$ were used, where the desired output is that x will have a coefficient of 1 and $\sin(x)$ will have a coefficient of 5. In this situation, the exact 'correct' terms are known, but in a real life apllication, many more terms with different frequencies and offsets of sin and cos could be fitted and then removed using regularisation. Due to the imperfect fit of the neural network, it is unlikely that the desired coefficients will be selected exactly, so other coefficients will try to fix any phase problems.

> **To-do:** *Expand on why I chose these basis functions.*

Table **??** shows a summary of the fit of the regression model, where there are many unwanted coefficients close to zero, but also many others far from zero. This model requires regularisation to reduce the number of terms and to remove the terms with coefficients very close to zero.

3.2.1 Stepwise regression

One way to regularising the model is to use stepwise regression to reduce the number of parameters by adding or removing one at a time and comparing the fit of the hierarchical models. For this scenario, it makes sense to use forward selection, where the initial model is empty and a term is added at each step, rather than backward selection, which starts with a full model and removes a term at each step. This is because we're trying to pick out the imprtant terms from a big lit of possibilities rather than remove a few

unimportant terms. In this case, the variables were selected using the Akaike information criterion (AIC), which penalises the fit for having a higher number of coefficients. The results of this regularisation are seen in Table 3.1.

The coefficients are slightly different, as seen in Table 3.3.

Table 3.1: The results of the model reduced using stepwise AIC.

Term	Estimate	Std. err.	t-value	p-value
x	1.034	0.002	484.452	0.000
$\sin(x)$	4.755	0.026	182.996	0.000
$\cos(x/2)$	0.292	0.025	11.461	0.000
$\cos(x)$	0.142	0.026	5.363	0.000
$\sin(x/2)$	0.122	0.027	4.558	0.000
$\sin(2x)$	0.042	0.026	1.622	0.105

While the relevant estimators x and $\sin(x)$ were identified and their coefficients fairly accurately estimated, a few other variables were also identified as significant. This method is only likely to work with a perfect or near perfect fit with no noise, which is unrealistic for real applications.

3.2.2 LASSO

Alternatively, we can use least absolute shrinkage and selection operator (LASSO) regularisation to select the significant variables from the full model. The LASSO constrains the sum of the absolute values of the model parameters, regularising the least influential parameters to zero. We vary the hyperparameter λ to change the trade off between simplicity and fit accuracy, as seen in Figure 3.1. We then use leave-one-out cross-validation to find the optimal hyperparameter λ, the results of which can be seen in Table 3.2.

To-do: *fix the discrepancy between the table and the figure (by adding a tolerance to the figure)*

3.2.3 Comparison

Table 3.3 shows a comparison between the coefficients selected by the full model and the two regularisation methods.

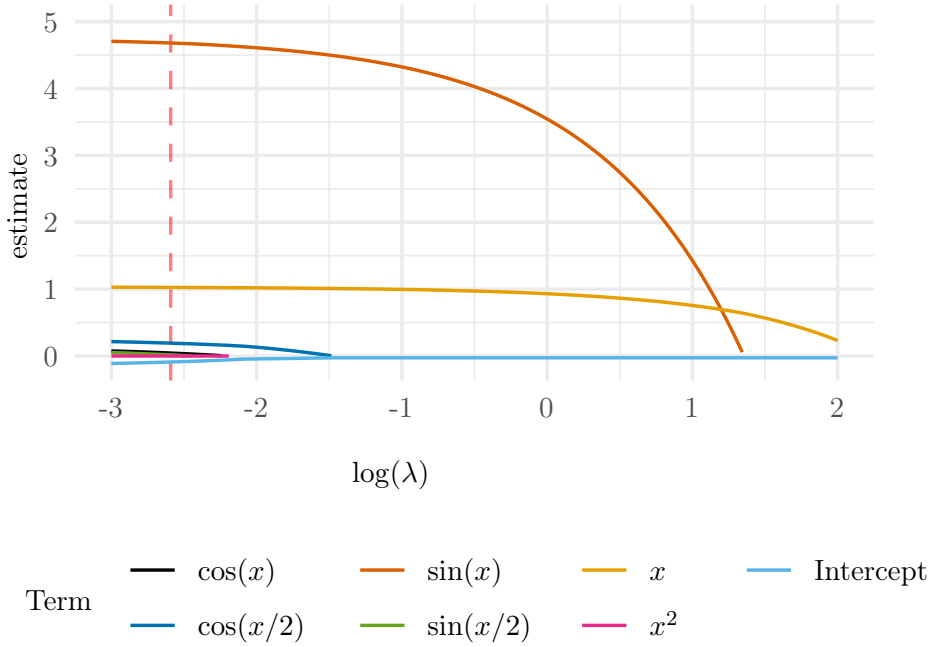

Figure 3.1: The effect on the coefficients when changing the regularisation hyperparameter λ.

Table 3.2: The optimised coefficients using leave-one-out cross-validation on the value of λ.

Term	Estimate
Intercept	-0.089
x	1.026
x^2	0.000
$\sin(x)$	4.681
$\sin(x/2)$	0.017
$\cos(x)$	0.044
$\cos(x/2)$	0.192

Table 3.3: The coefficients of the three regression models

Term	Multiple regression	Stepwise regression	LASSO
Intercept	-0.154		-0.089
x	1.034	1.034	1.026
x^2	0.001		0.000
$\sin(x)$	4.755	4.755	4.681
$\sin(2x)$	0.042	0.042	
$\sin(x/2)$	0.122	0.122	0.017
$\cos(x)$	0.141	0.142	0.044
$\cos(2x)$	-0.028		
$\cos(x/2)$	0.261	0.292	0.192

3.3 Gaussian processes

One proposed way to better understand deep learning is by using GPs to approximate the output of a deep learning algorithm. It has been shown that as the number of neurons in a layer of a neural network approaches infinity, the fit of the layer approaches a GP (Neal 1996).

In a similar way to how a univariate normal distribution with a mean and variance can be generalised to a multivariate normal distribution with a mean vector and a normal vector, a GP is the limit of extending a multivariate normal distribution to infinite dimensions, with a mean function and covariance function (sometimes also called a kernel in machine learning environments) which depends on the distance between two points.

A random vector $X \in \mathbb{R}^n$ is distributed with a multivariate normal dis-

tribution in n dimensions with mean vector $\boldsymbol{\mu} \in \mathbb{R}^n$ and covariance matrix $K \in \mathbb{R}^{n \times n}$ if $X \sim \mathcal{N}(\boldsymbol{\mu}, K)$. In a similar way, Y is a GP with a mean function μ and a covariance function k where $k(x_1, x_2) = k(|x_1 - x_2|)$ if $Y \sim \mathcal{GP}(\mu, k)$. Because a GP is an extension of a multivariate normal distribution, any finite subset of points from a GP has a multivariate normal distribution (Williams and Rasmussen 1996, p. 515).

To-do: *Explain how a GP will help us understand the neural network.*
GPs have statistical properties that we understand, so if we can fit a GP and then calculate these properties we can apply our knowledge to the neural network.

3.3.1 Using Gaussian process regression

Because GPs are very flexible, applying a GP to the output of the neural network is likely to result in a close fit.

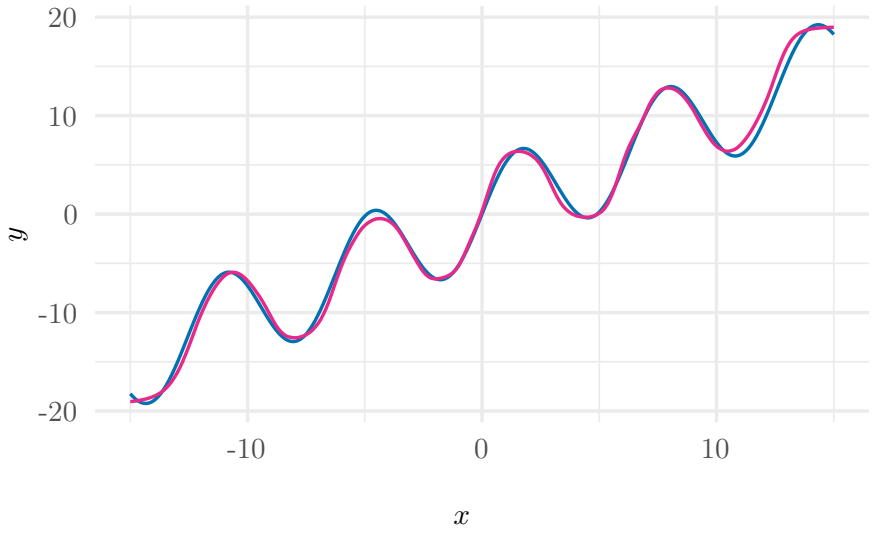

Figure 3.2: The fit of the GP (pink) and the true function (blue).

The fit of the GP is almost perfect, as expected.

To-do: *give some sort of quantitative explanation of how well the GP has fitted*
explain how the GP can now be used

Chapter 4

Conclusion

You can use these methods to:
Approximate the output of the neural network,
Put the output of the neural network in human understandable terms,
Find which terms are significant in the neural network model.

If you're trying to discover something about the real world, these techniques all rely on the neural network accurately reflecting reality which they probably don't. The fit of the statistical method will only be as good as the fit of the neural network, these techniques cannot diagnose a bad neural network.

4.1 How well have each of the attempts worked?

The stepwise regression was not as good as LASSO regularisation, which worked better. The GP was good, as expected.

It is not clear how well these techniques would transfer to higher dimensions and more complex situations as the GP took a long time to fit, and the regression didn't fit that well on a simple example.

4.2 What could be improved?

To-do: *Can we use Fourier transforms for seasonal data?*

4.3 How useful would more research on this topic be?

4.4 Future research

These techniques rely on the fit of the neural network to recover the original information, and the neural network could definitely fit better.

If using this, it would be sensible to first perform sensitivity analysis to find the significant variables.

For more complex applications, the use of GPs could be extended to deep (also known as hierarchical) GPs, which are analogous to deep neural networks (see Section 2.2). They involve chaining the output of one GP into another to better model nonlinearity (Damianou and Lawrence 2013).

One simple possibility to extend the models proposed would be to use multiple regression to find a trend and then use this as a mean function in a GP. This should allow the regression to capture most of the pattern and then the GP will provide a flexible way to capture the residuals.

4.5 Conclusion's conclusion

Deep learning is fast becoming a ubiquitous tool in many aspects of modern life — sometimes clearly visible, as with driverless cars, but in some cases more discreetly, such as the use of machine learning algorithms in American courts. As deep neural networks were designed to mimic a biological brain, a phenomenon that is still not well understood, they are a 'black box' whose reasoning is impossible to understand. Statistical methods such as GPs may offer a way to look inside this black box, as they offer a similar flexibility and wide range of uses, and are much more easily interpreted by humans. So far, much of the work that has been done involving GPs and machine learning has been comparative, rather than using one to model the other.

Appendix A

Code

Bibliography

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M. *et al.* (2016). 'TensorFlow: A system for large-scale machine learning'. 16, pp. 265–283. arXiv: `http://arxiv.org/abs/1605.08695v2 [cs.DC]`.

Allaire, J. J. and Chollet, F. (2018). *keras: R Interface to 'Keras'*. R package version 2.2.0. URL: `https://CRAN.R-project.org/package=keras`.

Bengio, Y., Louradour, J., Collobert, R. and Weston, J. (2009). 'Curriculum learning'. *Proceedings of the 26th annual international conference on machine learning*. ACM, pp. 41–48.

Chao, X., Kou, G., Li, T. and Peng, Y. (2018). 'Jie Ke versus AlphaGo: A ranking approach using decision making method for large-scale data with incomplete information'. *European Journal of Operational Research* 265.1, pp. 239–247. DOI: `10.1016/j.ejor.2017.07.030`.

Chollet, F. *et al.* (2015). *Keras*. `https://keras.io`.

Christin, A., Rosenblat, A. and Boyd, D. (2015). 'Courts and predictive algorithms'. *Data & CivilRight*.

Cortez, P. and Embrechts, M. J. (2013). 'Using sensitivity analysis and visualization techniques to open black box data mining models'. *Information Sciences* 225, pp. 1–17. DOI: `10.1016/j.ins.2012.10.039`.

Damianou, A. and Lawrence, N. (2013). 'Deep gaussian processes'. *Artificial Intelligence and Statistics*, pp. 207–215.

Gerla, M., Lee, E., Pau, G. and Lee, U. (2014). 'Internet of vehicles: From intelligent grid to autonomous cars and vehicular clouds'. *2014 IEEE World Forum on Internet of Things (WF-IoT)*. IEEE, pp. 241–246. DOI: `10.1109/WF-IoT.2014.6803166`.

Herbrich, R., Lawrence, N. D. and Seeger, M. (2003). 'Fast sparse Gaussian process methods: The informative vector machine'. *Advances in neural information processing systems*, pp. 625–632.

LeCun, Y., Bengio, Y. and Hinton, G. (2015). 'Deep learning'. *Nature* 521.7553, pp. 436–444. DOI: 10.1038/nature14539.

Li, T., Katz, R. H. and Culler, D. E. (2018). 'ConNect: Exploring Augmented Reality Service using Image Localization and Neural Network Object Detection'.

Linnainmaa, S. (1970). 'Alogritmin kumulatiivinen pyöristysvirhe yksittäisten pyöristysvirheiden Taylor-kehitelmänä (The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors)'. Master's Thesis. University of Helsinki, pp. 6–7.

MacKay, D. J. C. (1997). 'Gaussian processes: a replacement for supervised neural networks?'

Mahendran, A. and Vedaldi, A. (2014). 'Understanding Deep Image Representations by Inverting Them'. arXiv: http://arxiv.org/abs/1412.0035v1 [cs.CV].

Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. MIT Press Ltd. 1104 pp. ISBN: 0262018020. URL: https://www.ebook.de/de/product/19071158/kevin_p_murphy_machine_learning.html.

Neal, R. M. (1996). 'Priors for Infinite Networks'. *Bayesian Learning for Neural Networks*. Springer New York, pp. 29–53. DOI: 10.1007/978-1-4612-0745-0_2.

Ramachandran, P., Zoph, B. and Le, Q. V. (2017). 'Searching for Activation Functions'. arXiv: http://arxiv.org/abs/1710.05941v2 [cs.NE].

Rasmussen, C. E. (1997). *Evaluation of Gaussian processes and other methods for non-linear regression*. University of Toronto.

Rosenblatt, F. (1957). *The perceptron, a perceiving and recognizing automaton*. Cornell Aeronautical Laboratory.

— (1958). 'The perceptron: A probabilistic model for information storage and organization in the brain.' *Psychological Review* 65.6, pp. 386–408. DOI: 10.1037/h0042519.

Schmidhuber, J. (2015). 'Deep learning in neural networks: An overview'. *Neural Networks* 61, pp. 85–117. DOI: 10.1016/j.neunet.2014.09.003.

Thoma, M. (2017). 'Analysis and Optimization of Convolutional Neural Network Architectures'. Master's Thesis. Karlsruhe Institute of Technology. arXiv: http://arxiv.org/abs/1707.09725v1 [cs.CV]. URL: https://martin-thoma.com/msthesis/.

Werbos, P. J. (1982). 'Applications of advances in nonlinear sensitivity analysis'. *System modeling and optimization*. Springer, pp. 762–770.

Williams, C. K. I. and Rasmussen, C. E. (1996). 'Gaussian Processes for Regression'. *Advances in neural information processing systems*, pp. 514–520.