University of Exeter

Can statistics help us to understand deep learning?

Johannes Smit

May 2019

Abstract

Machine learning and deep neural networks have seen widespread success in many applications and are set to play a larger rôle in modern society — sometimes visibly, as with driverless cars, but in some cases more discreetly, such as the use of machine learning algorithms in the U.S. judicial system. Due to their hierarchical structure, deep neural networks are very difficult or impossible for humans to understand, which could cause problems when a machine learning algorithm does something unforeseen when in a position of power. This project uses a simple nonlinear function as an example to train a deep neural network and then applies multiple regression and Gaussian processes to the output of the neural network to try and extract human understandable meaning. LASSO regularisation is used to reduce the regression model to a more human understandable form and is then combined with a Gaussian process to fit the neural network better.

Acknowledgements

Thank you to my supervisor Peter Challenor for his expertise and guidance, to my family for their constant support throughout my time at university, to Tom for his friendship these last four years and to Alison for her dedication. All in all, thanks be to God.

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.

This project 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 describe some properties of the neural network. This will involve fitting a deep neural network to a known function, then using the neural network to generate a new dataset that approximates the training set. Then, a series of multiple regression models will be fitted to the output of the neural network to try and recover the form of the original formula. A Gaussian processes will also be fitted to the output of the neural network, which can then be analysed using the robust framework of Gaussian processes in an attempt to describe properties of the neural network's fit.

Chapter 2

Machine learning

The sudden rise in success of machine learning is largely due 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 hierarchical models such as deep neural networks to be trained on large amounts of data in a reasonable amount time.

2.1 Artificial neural networks

One of the most popular types of machine learning algorithm is the artificial neural network, a parametric model that has seen widespread success 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 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 objection to 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). He also wanted his synthetic brain to mirror biological brains' redundancy, the ability to have pieces removed and still function. This 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 result of his work was the 'perceptron', a machine 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).

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

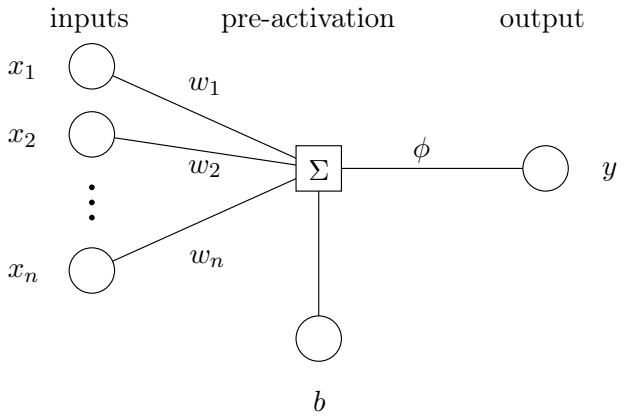


Figure 2.1: A diagram of an example artificial neuron.

2.1.3 Activation functions

Early examples of artificial neurons were built in hardware, where the output would be a bulb 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 other neurons, as information about the magnitude of the output is lost. Commonly, the sigmoid function, defined as $S(x) = 1/(1 + e^{-x})$; the 'softmax' function, a generalisation of the logistic function to higher dimensions or the tanh function are used. 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 network, 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.

An efficient method to optimise (or 'train') the weights in a neural

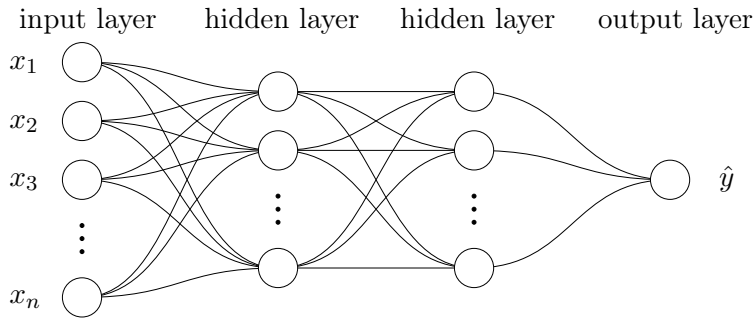


Figure 2.2: An example of the structure of a fully connected feedforward neural network.

network was not known until Linnainmaa (1970) published his method of automatic differentiation, which was first applied to neural networks by Werbos (1982). To perform the backpropagation algorithm, 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. The neural network's weights must be altered to reduce this error, 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 starting at the output layer where the error is known and 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 produce the largest decrease in the error.

7. Update the weights $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta\boldsymbol{\Delta}$, where η is the step size hyperparameter, called the 'learning rate', 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 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:

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

This means each epoch consists of only one step, but this step takes account of all the datapoints, and so will converge in fewer epochs, however each epoch takes so long to compute that this takes significantly longer to train. In practice, 'minibatches' are used, where the dataset is split into small groups, normally 32 datapoints or fewer, which 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 being affected by 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 some mixed success in the 20th century, when the number of layers was increased, the resulting neural network was impractical to train with the techniques and hardware of the time. It was only in the 1990s that training these deep neural networks became feasible, leading to the widespread success of deep learning 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 will not always improve the model. As neural networks get deeper, they encounter the 'vanishing gradient problem'. This is where the gradient at each layer in the backpropagation algorithm gets smaller until there is almost no change to the weights near the input. 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, AI voice assistants are becoming more common in homes and self-driving cars are starting to become commercially available, all of which use some form of machine learning. 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 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 function was chosen so that a deep neural net would easily fit it well, and then any attempts at understanding the neural network can be easily compared to the 'true' function. The function only has one input and one output so that the results can be plotted on a 2-dimensional plane. 256 evenly spread datapoints were drawn from this function, seen in Figure 2.3.

2.3.1 Training a neural network

The Keras package for R (Allaire and Chollet 2018) was used to create and train a neural network on the dataset in Figure 2.3.

The number of hidden layers in the neural network was slowly 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 chosen as the most suitable for this task after being compared with the ReLU, sigmoid and softmax activation functions, (see Section 2.1.3 for a description of each of these functions). A

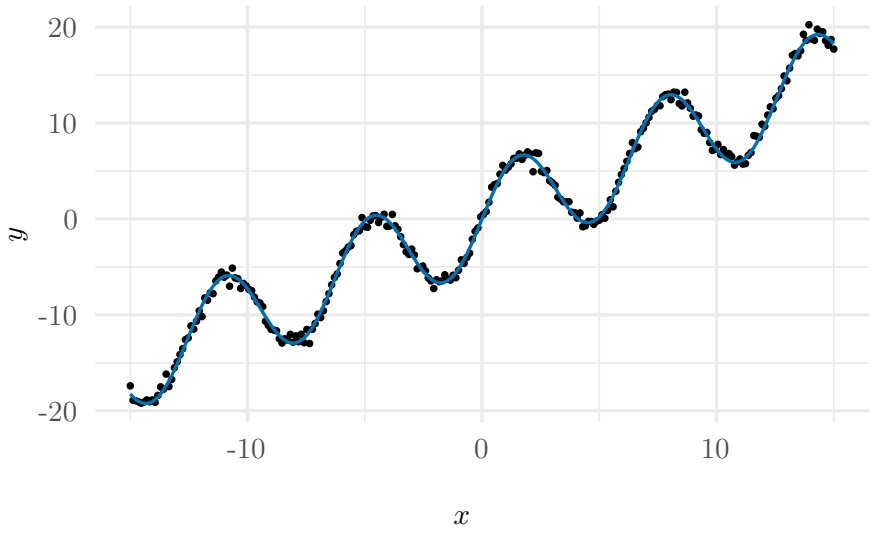


Figure 2.3: The true function $f(x) = x + 5\sin(x)$ (blue) and the 256 training points (black).

linear activation function was required for the 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. This structure has 801 parameters to optimise, the 720 weights seen in Figure 2.4 and 81 biases.

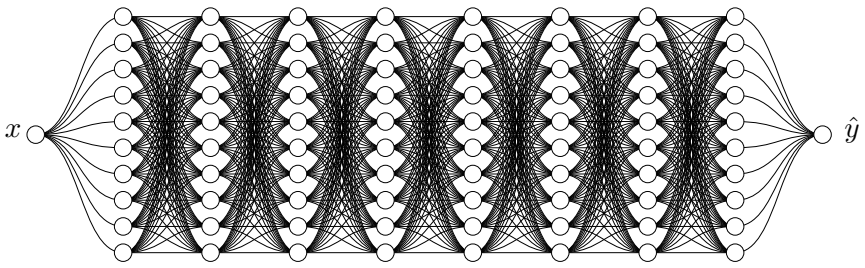


Figure 2.4: The final architecture.

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, so minibatches were chosen to allow the algorithm to more easily get out of valleys (see Section 2.3.2). The neural net was trained for 2,000 epochs to ensure that the weights had stably converged and the training had finished. This can be seen in Figure 2.5, where the mean squared error stops improving or improves very slowly after a few hundred epochs. The final result of this training process is seen in Figure 2.6.

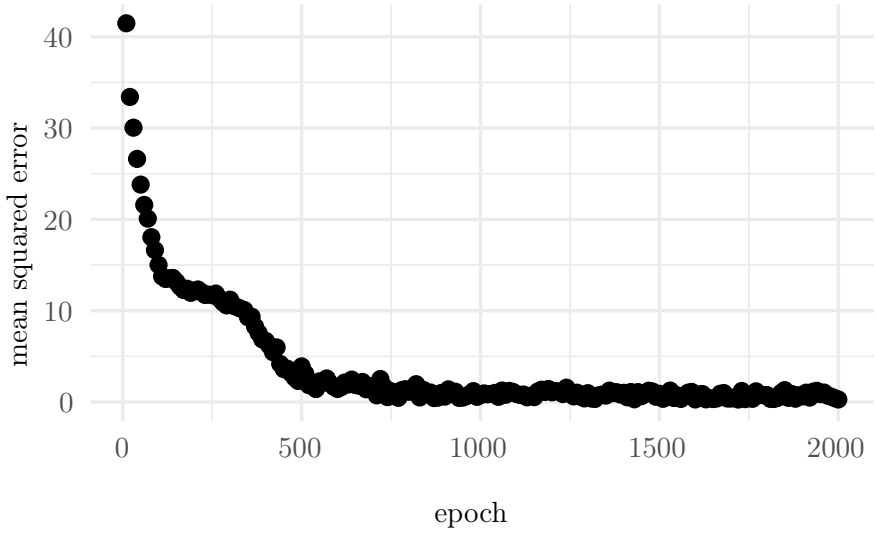


Figure 2.5: The mean squared error at different epochs.

2.3.2 Ordering of the datapoints

The training dataset was constructed with the x values increasing. However, when the neural network was first trained on this dataset, the algorithm seemed to always get stuck in a local minimum, as seen in Figure 2.7a, where the fit on the right-hand side is very bad. When the order of the datapoints was reversed, i.e., the dataset was ordered with x decreasing, the opposite side of the data fitted badly, see Figure 2.7b.

Due to the high dimensional weight space, the gradient descent algorithm must traverse a surface that has many areas with zero gradient that ae difficult to escape. Using minibatches reduces the likelihood of getting stuck because each minibatch has a different error surface, so the algorithm will take a step in a slightly different direction for each minibatch. This allows the algorithm to escape a valley in one error surface by taking a step downhill in another error surface. However, if the datapoints are not sorted before being fitted, each minibatch will contain a highly correlated subset of the

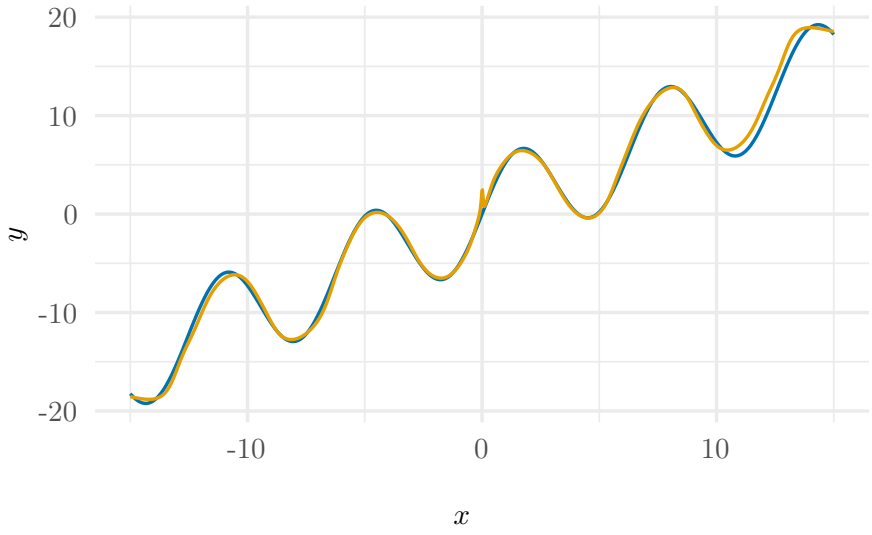


Figure 2.6: The true values (blue) and the neural network's predicted values (yellow).

overall dataset, and will therefore not be representative of the rest of the data. This can lead to the steps that the gradient descent algorithm takes in an epoch counteracting each other, which can stop any changes in the weights from being made.

I believed that this was what was causing the bad fit seen in Figures 2.7a and 2.7b. It seemed that the datapoints in the first batches had the largest effect on the training of the model, and whichever batch was last to be inputted had the least effect on the fit. To counteract this, I randomly shuffled the datapoints before using them to train the neural network. This resulted in the significantly better fit seen in Figure 2.7c. I also tried deterministically separating the datapoints so that the datapoint with index i was sorted into minibatch $m = i \pmod 8$, where $m = 0, \ldots, 7$. This way each minibatch would contain an even spread of datapoints. This deterministic approach also resulted in a good fit, seen in Figure 2.7d.

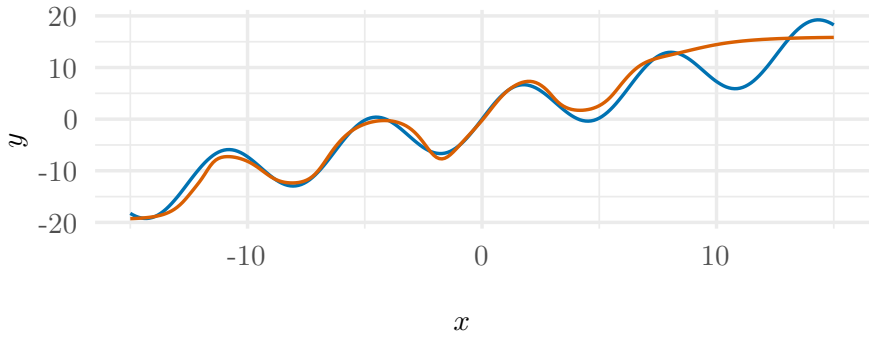
However, when the neural network was trained using batch gradient descent, this problem did not go away. Batch gradient descent calculates the error for all the datapoints at once, so should not be affected by the order of the datapoints. This lead me to discover that the problem was actually to do with how I was using an 80%/20% training/validation split to cross-validate the neural network. My mistake was assuming that Keras was choosing a random sample of datapoints for each subset, when it was actually choosing the first 80% for the training data and holding back the remaining 20% for

the validation set. This meant that when the datapoints were sorted by ascending x, the right-hand side was in the validation set, and so wasn't used for training. Similarly, when reversed, the left-hand side was in the validation set. Shuffling and deterministically separating the datapoints meant that the validation set was a representative sample of the dataset, and so did not bias the model.

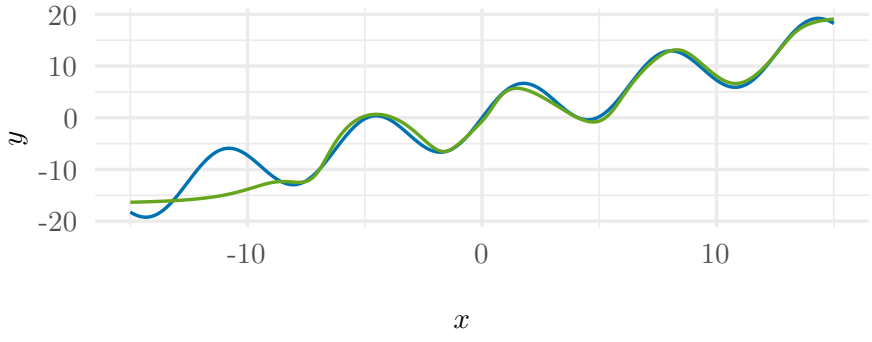
Once the validation split was set to 0%, all of the ordering methods produced similar results. The example neural network model seen in Figure 2.6 and used from now on is one trained with this issue fixed. However, there is still a small spike at $x = 0$ that I do not know how to fix.

Bengio *et al.* (2009) developed a technique called 'curriculum learning' which uses the ordering of the datapoints 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.

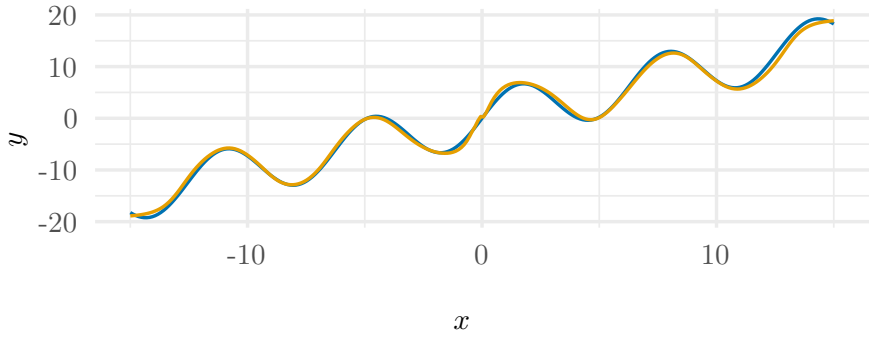
For example, when trained on a text dataset, the algorithm was trained on only texts using the vocabulary of 5,000 most common words. The vocabulary was expanded by 5,000 words at intervals, allowing the algorithm to learn new words once it had mastered simpler words. When compared with an algorithm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



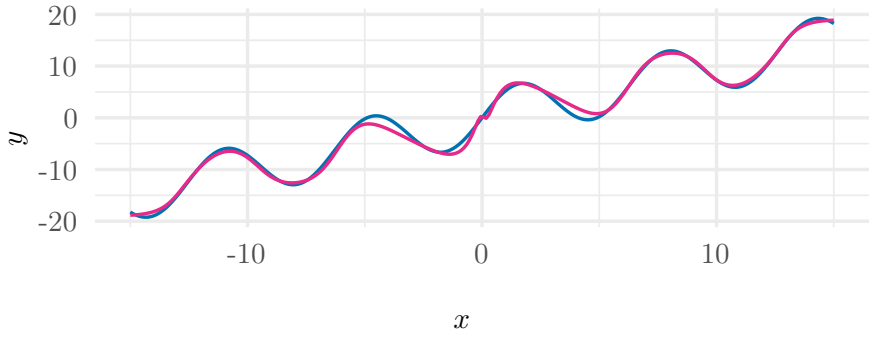
(a) Ascending x



(b) Descending x



(c) Shuffled



(d) Deterministic

Figure 2.7: The output of the same neural network trained on the same data but ordered differently: with x ascending (orange), with $x$descending (green), shuffled (yellow) and deterministically separated (pink) compared to the true function (blue).

Chapter 3

Opening the black box

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. Directly understanding the meaning of each weight in a neural network is impossible for a human. Even the neural network used to learn the simple example in this project seen in Figure 2.4 is far too complex to keep track of.

3.1 Current and past research

When neural networks were first 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, each neuron is activated by a seemingly random set of pixels spread across the image. For this reason, it is very difficult to give meaning to any neuron in an image recognising neural network, unlike what was originally hypothesised.

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).

3.2 Multiple regression

An ideal way to understand a deep neural network would be to find the form of a function that describes the output of the neural network. One way of doing this is to use multiple regression with a series of basis functions on the output of the neural network. A new dataset is generated by using the neural network to predict the target values in the domain that it was trained on. Then 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 that the neural network is trying to approximate. 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.

In this case, 1024 datapoints were generated from the neural network, and the basis functions 1, x, x^2, $\sin(x)$, $\sin(2x)$, $\sin(x/2)$, $\cos(x)$, $\cos(2x)$ and $\cos(x/2)$ were used. The ideal fit for the model is that x and $\sin(x)$ will be selected as significant and ideally have coefficients close to 1 and 5 respectively and the other terms will have coefficients close to zero. In this situation, the exact 'true' terms are known, but in a more realistic application where the form of the input function is completely unknown, many more terms could be fitted including higher order polynomials and interactions between predictors. Due to the imperfect fit of the neural network, it is unlikely that the desired coefficients will be selected exactly, so other terms will have non-zero coefficients to slightly adjust the model's fit.

Table 3.1 shows a summary of the fit of the regression model. There are many terms with coefficients close to zero which could be ignored as they do not contribute much to the fit, although almost all of the terms have t-values in the extreme tails of the distribution.

This model requires regularisation to reduce the number of terms and to determine which of the insignificant terms to remove.

3.2.1 Stepwise regression

One way to regularise the model is to use stepwise regression to reduce the number of parameters. The algorithm works by adding or removing one

Table 3.1: The results of the multiple regression model.

Term	Estimate	Std. err.	t-value	p-value
Intercept	0.055	0.023	2.405	0.016
x	1.034	0.002	597.727	0.000
x^2	0.000	0.000	1.493	0.136
$\sin(x)$	4.811	0.021	228.482	0.000
$\sin(2x)$	0.030	0.021	1.460	0.145
$\sin(x/2)$	0.008	0.022	0.352	0.725
$\cos(x)$	0.114	0.022	5.296	0.000
$\cos(2x)$	0.086	0.021	4.063	0.000
$\cos(x/2)$	0.128	0.023	5.553	0.000

term at a time and comparing the fit of the nested models. In this case, the variables were selected using the Akaike information criterion (AIC), which penalises the fit for having a higher number of coefficients. For this example, forward selection — where the initial model is empty and a term is added at each step, and backward selection — which starts with a full model and removes a term at each step, were used. Both arrived at similar models, but the method with the lowest AIC was to use both forward and backward selection. Table 3.2 shows a summary of the reduced model. The intercept and $\cos(2x)$ terms were removed from the full model. Some of the coefficients in the reduced model are slightly different from the full model to compensate for the removed terms, as seen in Table 3.4.

Table 3.2: The summary of the model reduced using stepwise AIC.

Term	Estimate	Std. err.	t-value	p-value
x	1.034	0.002	599.157	0.000
$\sin(x)$	4.812	0.021	231.087	0.000
$\cos(x/2)$	0.116	0.023	5.142	0.000
$\cos(x)$	0.113	0.022	5.224	0.000
x^2	0.001	0.000	4.982	0.000
$\cos(2x)$	0.089	0.021	4.185	0.000
$\sin(2x)$	0.030	0.021	1.449	0.148

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.

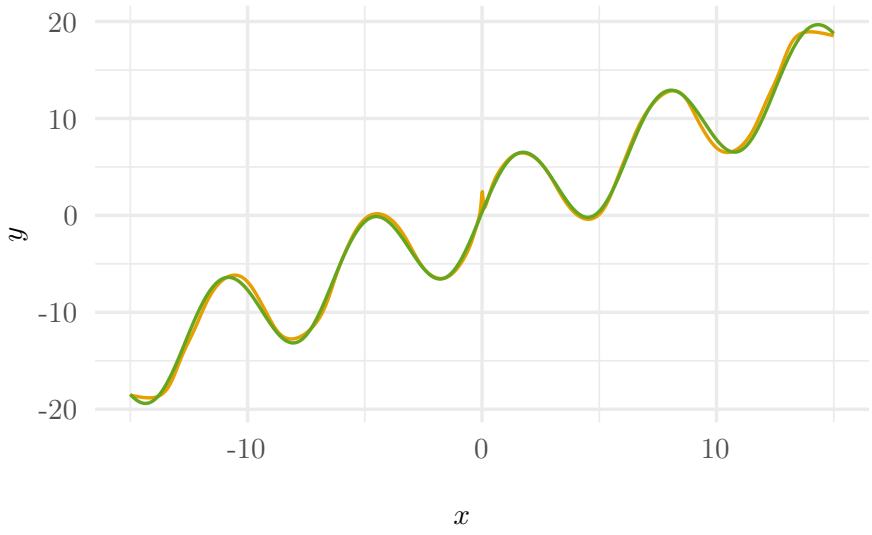


Figure 3.1: The fit of the stepwise model (green) and the output of the neural network (yellow).

3.2.2 LASSO

A more modern alternative to stepwise regression is least absolute shrinkage and selection operator (LASSO) regularisation. The LASSO constrains the sum of the absolute values of the model parameters, regularising the least influential parameters to zero. The hyperparameter λ controls the trade off between model simplicity and fit accuracy, as seen in Figure 3.2. Then leave-one-out cross-validation is used to find the optimal value for λ, the results of which can be seen in Table 3.3.

Table 3.3: The optimised coefficients using leave-one-out cross-validation on the value of λ.

Term	Estimate
Intercept	0.094
x	1.026
$\sin(x)$	4.721
$\cos(x)$	0.024
$\cos(x/2)$	0.045

The LASSO has removed many more terms than the stepwise regression, as seen in Table 3.4.

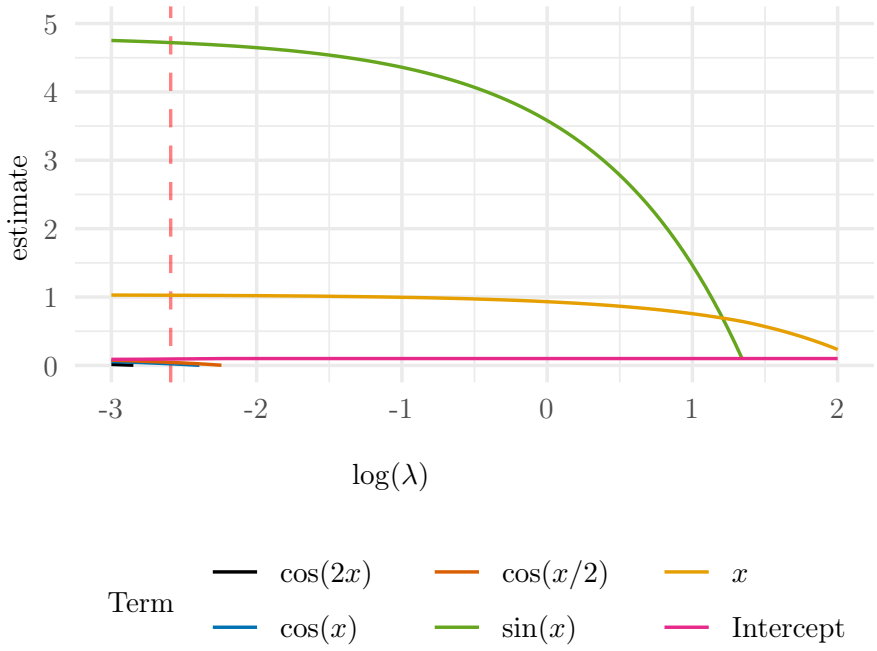


Figure 3.2: The effect on the coefficients when changing the regularisation hyperparameter λ and the optimised value of λ (dashed red).

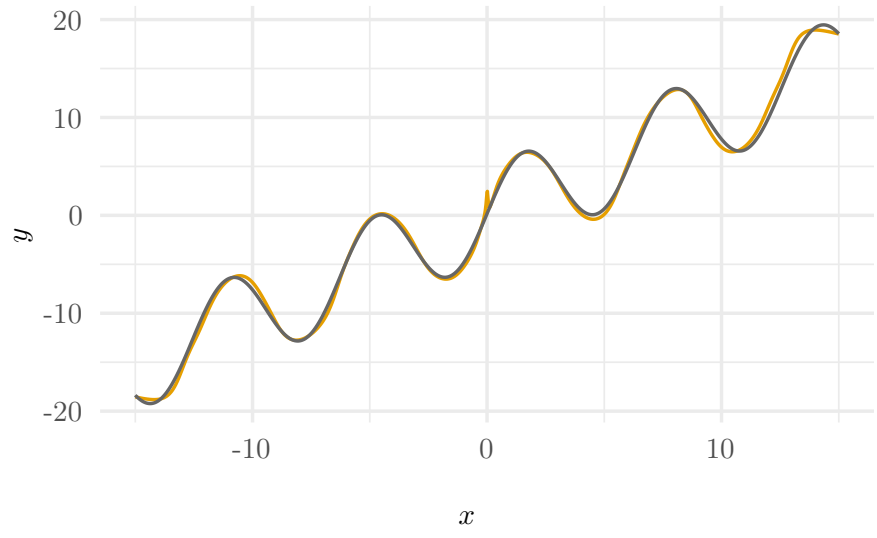


Figure 3.3: The fit of the LASSO (grey) and the output of the neural network (yellow).

3.2.3 Comparison of linear regression models

Table 3.4 shows a comparison between the coefficients selected by the full model and the two regularised models. The LASSO model removed many more terms that had coefficients close to zero than the stepwise regression model, and seems to have shown that the form of the equation for the neural network consists mainly of x and $\sin(x)$ terms, with a small negative intercept. The $\sin(x/2)$ coefficient is so small that it makes almost no difference to the model. Interpreting the stepwise regression model is slightly more difficult as there are more terms. The x and $\sin(x)$ terms make up the majority of the fit, and most other coefficients are close to zero, leaving only a few phase shifting terms such as $\sin(x/2)$.

Table 3.4: The coefficients of the three linear regression models.

Term	Multiple regression	Stepwise regression	LASSO
Intercept	0.055		0.094
x	1.034	1.034	1.026
x^2	0.000	0.001	
$\sin(x)$	4.811	4.812	4.721
$\sin(2x)$	0.030	0.030	
$\sin(x/2)$	0.008		
$\cos(x)$	0.114	0.113	0.024
$\cos(2x)$	0.086	0.089	
$\cos(x/2)$	0.128	0.116	0.045

3.3 Gaussian processes

A nonparametric alternative to multiple regression is the Gaussian process (GP), which can also be used to approximate the output of a deep learning algorithm. One reason this might be successful is because it has been shown that the fit of a neuron tends to a GP as the number of inputs approaches infinity (Neal 1996).

Similar 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 distribution 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).

For a more in-depth discussion of GPs, see the literature review (Smit 2018).

3.3.1 Using Gaussian process regression

There appears to be no literature on using 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. The most commonly cited advantages of GPs are faster convergence and the ability to calculate confidence intervals (or credible intervals) for predictions, which neural networks cannot do (Herbrich, Lawrence and Seeger 2003).

A GP can be fitted to the output of a neural network in the same way as the multiple regression model in Section 3.2. GPs have well understood statistical properties such as a length-scale and robust sensitivity analysis techniques can be applied to the fit of the neural network to provide another human understandable perspective to the neural network. In this example, a GP was fitted to a smaller set of 256 datapoints produced by the neural network, as the better fit when using the full 1024 points did not warrant the significantly longer fitting time[1]. The GP was fitted with a squared exponential covariance function and no nugget, so that it would interpolate between the points rather than smooth them, although there are likely contexts where some smoothing would be useful. Because GPs are very flexible, applying a GP to the output of the neural network is likely to result in a close fit.

[1]Software: R 3.5.3, 'mlegp' package (Dancik and Dorman 2008); Hardware: Intel Core i5-4210U processor, 8GB RAM

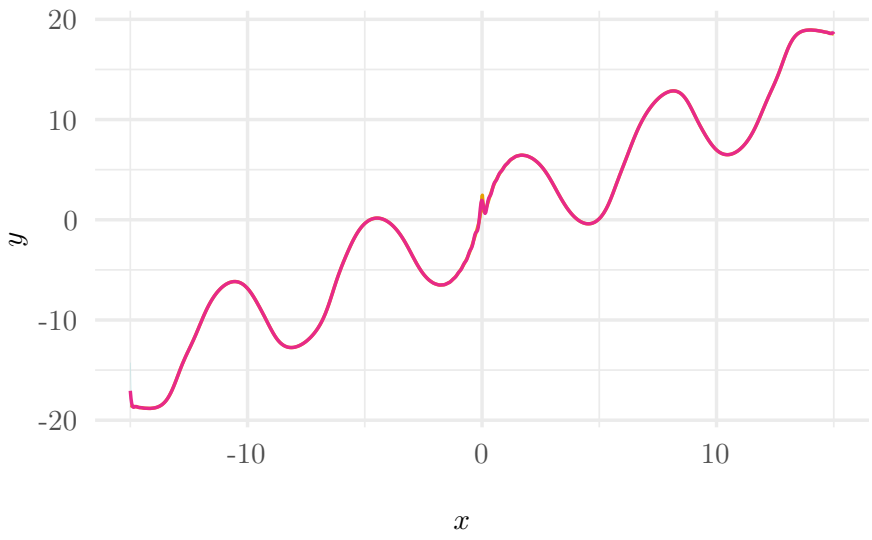


Figure 3.4: The fit of the GP and 99% confidence interval (pink) and the neural network (yellow). The confidence interval is so narrow that it is hidden and the fit of the GP is so close to the neural network's fit that it is almost impossible to see.

Figure 3.4 shows that the fit of the GP to the neural network is almost perfect.

3.4 Combining regression and Gaussian processes

The fit of the LASSO model is not perfect, as seen in Figure 3.3. One possible way to improve the fit of this model is to use a GP to capture the residuals of the LASSO model. The residuals of the model are seen in Figure 3.5, where a GP has been fitted.

Figure 3.6 shows the GP added to the LASSO model to capture the error. Judging by eye, the fit is close to perfect, similar to using a GP on its own. However, this method has the advantage of providing information about the form of the equation for the neural network's output though the LASSO component and also some information about the residuals captured by the GP.

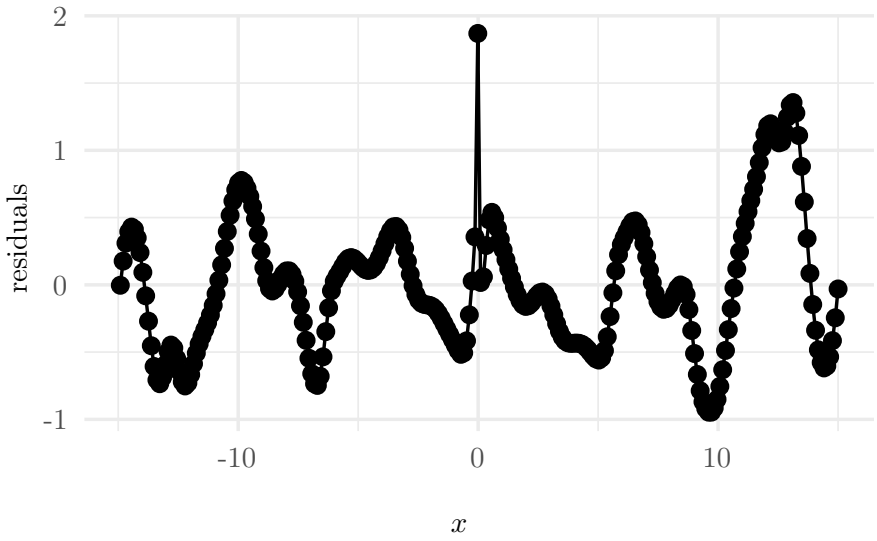


Figure 3.5: The fit of the GP to the residuals of the LASSO model.

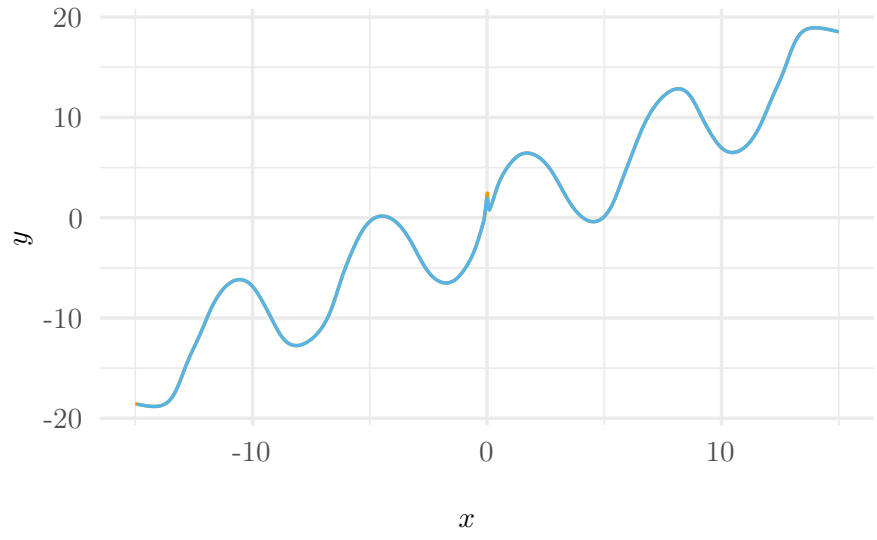


Figure 3.6: The fit of the combined LASSO and GP model (light blue) and the neural network (yellow).

Chapter 4

Conclusion

The methods discussed in this project have been successfully fitted to the output of a deep neural network and used to learn more about its properties. The form of the equation used to train the neural network was recovered by fitting a linear model with a series of basis functions. Regularisation techniques allowed the model to be simplified further to only a few interpretable terms. The LASSO worked much better than stepwise regression to select a few important terms and also allows for more fine control of the regularisation strength. GPs also were used to provide a nonparametric method to understand the neural network, and also managed to capture the residuals of the LASSO model.

These tests have relied on the fit of the neural network to be sufficiently good that the original function can be recovered. In realistic applications, the process that produced the data is unknown, so there is no way to check how accurate the regression's understanding is. Further testing with more complex training datasets would need to be done to see how well these techniques generalise to higher dimensions and more complex functions.

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. They involve chaining the output of one GP into another to better model nonlinearity (Damianou and Lawrence 2013).

Deep learning is fast becoming a widespread 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 the U.S. judicial system. As deep neural networks were designed to mimic biological brains which are 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

00-main.R

```
# set up ----
library(kableExtra) # for outputting LaTeX tables with kable()
library(MASS) # for stepwise regression
library(broom) # to tidy() model outputs
library(here) # to get relative file paths
library(glmnet) # for LASSO regularisation
library(mlegp) # for Gaussian process regression
library(tidyverse)
library(magrittr) # for pipe aliases e.g. use_series()
library(keras) # for ANNs

# set seed for R and Keras and disable other sources of randomness
use_session_with_seed(321)

source("01-helper-functions.R")

source("02-create-dataset.R")

# machine learning ----
load_neural_network <- TRUE
source("03-machine-learning.R")

# multiple regression ----
source("04-basis-functions.R")
source("05-multiple-regression.R")
source("06-stepwise-regression.R")
source("07-lasso.R")
source("08-compare-coefs.R")

# gaussian process ----
source("09-gaussian-process.R")

# regression and gp ----
source("10-regression-and-gp.R")
```

01-helper-functions.R

```r
# create a function to make a string into LaTeX inline math mode
math_mode <- function(x, format = "") {
  if (format == "$") {
    paste0("$", x, "$")
  } else {
    paste0("\\(", x, "\\)")
  }
}

path_figures <- here("figures")
golden <- (1 + sqrt(5)) / 2
save_plot <- function(plot = last_plot(), filename, w = 10, aspect_ratio = golden) {
  ggsave(
    plot = plot,
    filename = paste0(filename, ".svg"),
    path = path_figures,
    width = w,
    height = w / aspect_ratio,
    units = "cm"
  )
}

# set up tables
path_tables <- here("tables")
get_model_summary <- function(model) {
  column_names <- list(
    "Term" = "latex",
    "Estimate" = "estimate",
    "Std. err." = "std.error",
    "\\(t\\)-value" = "statistic",
    "\\(p\\)-value" = "p.value"
  )
  model %>%
    tidy() %>%
    inner_join(basis_functions_latex, by = c("term" = "name")) %>%
    select(!!!column_names)
}

options(knitr.kable.NA = "") # remove NAs from exported tables

# create a function to format numbers in scientific notation
scientific <- function(x) {
  p <- floor(log10(abs(x)))
  if (abs(p) > getOption("scipen")) {
    b <- round(x / 10^p, 3)
    return(math_mode(paste0(b, " \\times 10^{", p, "}")))
  } else {
    return(as.character(x))
  }
}

save_table <- function(table, label = NA, caption = NA, digits = 3) {
  table %>%
```

```
    kable(
      format = "latex",
      label = label,
      caption = caption,
      digits = digits,
      linesep = "",
      escape = FALSE,
      booktabs = TRUE
    ) %>%
    kable_styling(latex_options = "hold_position") %>%
    cat(file = paste0(path_tables, label, ".tex"))
}

# set up ggplot
theme_set(
  theme_minimal() +
    theme(
      axis.title.x = element_text(margin = margin(t = 10)),
      legend.position = "none"
    )
)

# create a function to add cartesian axis labels
cartesian_labels <- function() {
  labs(x = "$x$", y = "$y$")
}

# create a custom colour scheme
unnamed_colours <- c(
  "#000000",
  "#0072B2",
  "#D95F02",
  "#66A61E",
  "#E69F00",
  "#E7298A",
  "#56B4E9",
  "#F0E442",
  "#666666"
)

model_colours <- c(
  train = "#000000", # black
  truth = "#0072B2", # blue
  nn = "#E69F00", # yellow
  mr = "#D95F02", # orange
  sr = "#66A61E", # green
  lasso = "#666666", # grey
  gp = "#E7298A", # pink
  combined = "#56B4E9" # light blue
)

model_types <- names(model_colours)
```

02-create-dataset.R

```r
# create dataset ----

# create a nonlinear function
my_function <- function(x) {
  x + 5 * sin(x)
}

# create a training dataset
ml_train_n <- 32 * 8
ml_train_limits <- c(-15, 15)
ml_train_x <- seq(ml_train_limits[1], ml_train_limits[2], len = ml_train_n)

ml_truth_y <- my_function(ml_train_x)

ml_train_df <- tibble(
  x = ml_train_x,
  y = ml_truth_y + rnorm(ml_train_n, mean = 0, sd = 0.5),
  model = factor("train", levels = model_types)
)

# plot the dataset
gg_sin_x_dataset <- ggplot(data = ml_train_df, mapping = aes(x, y)) +
  geom_point(mapping = aes(colour = "train"), size = 0.25) +
  stat_function(fun = my_function, n = ml_train_n, mapping = aes(colour = "truth")) +
  scale_colour_manual(values = model_colours) +
  cartesian_labels()
save_plot(plot = gg_sin_x_dataset, filename = "sin-x-dataset")
```

03-machine-learning.R

```r
# machine learning ----

# set up training variables
n_epochs <- 2000

# shuffle training dataset
ml_train_df <- arrange(ml_train_df, sample(ml_train_n))

# create a blank neural network model
blank_model <- function() {
  keras_model_sequential() %>%
    layer_dense(units = 10, kernel_initializer = "RandomNormal",
                activation = "tanh", input_shape = 1) %>%
    layer_dense(units = 10, activation = "tanh") %>%
    layer_dense(units = 10, activation = "tanh") %>%
    layer_dense(units = 10, activation = "tanh") %>%
    layer_dense(units = 10, activation = "tanh") %>%
    layer_dense(units = 10, activation = "tanh") %>%
    layer_dense(units = 10, activation = "tanh") %>%
    layer_dense(units = 10, activation = "tanh") %>%
    layer_dense(units = 1, activation = "linear") %>%
    compile(
      loss = "mse",
      optimizer = optimizer_rmsprop(),
      metrics = list("mean_squared_error")
```

```
    )
}

model_filepath <- here("models", "neural-network.h5")
training_history_filepath <- here("models", "training-history.RData")
if (load_neural_network) {
  if (!file.exists(model_filepath)) {
    # error if file does not exist
    stop("Neural network file '", model_filepath, "' not found!")
  } else if (!file.exists(training_history_filepath)) {
    # error if file does not exist
    stop("Neural network file '", training_history_filepath, "' not found!")
  }else{
    # load the neural network from a file
    ml_model <- load_model_hdf5(model_filepath)
    load(training_history_filepath)
  }
} else {
  # fit the model
  message("Fitting the neural network")
  ml_model <- blank_model()
  ml_training_history <- fit(
    ml_model,
    x = ml_train_df$x,
    y = ml_train_df$y,
    batch_size = 32,
    epochs = n_epochs,
    shuffle = TRUE,
    verbose = 0
  )
  message("Neural network fitted")

  # save the neural network
  save_model_hdf5(ml_model, filepath = model_filepath)
  save(ml_training_history, file = training_history_filepath)

  # plot the training history
  gg_training_history <- tibble(
    epoch = seq(n_epochs),
    mse = ml_training_history$metrics$mean_squared_error
  ) %>%
    filter(epoch %% 10 == 0) %>%
    ggplot(mapping = aes(x = epoch, y = mse)) +
    geom_point(size = 2, stroke = 0) +
    expand_limits(y = 0) +
    labs(y = "mean squared error")
  save_plot(plot = gg_training_history, filename = "training-history")
}

# predict from the neural network
ml_pred_n <- 2^10
ml_pred_x <- seq(ml_train_limits[1], ml_train_limits[2], len = ml_pred_n)
ml_pred_y <- as.numeric(predict(ml_model, ml_pred_x))
ml_pred_df <- tibble(
  x = ml_pred_x,
```

```
  y = ml_pred_y,
  model = factor("nn", levels = model_types)
)

gg_nn_fit <- ggplot(data = ml_pred_df, mapping = aes(x, y, colour = model)) +
  stat_function(fun = my_function, n = ml_train_n, mapping = aes(colour = "truth")) +
  geom_line() +
  scale_colour_manual(values = model_colours) +
  cartesian_labels()
save_plot(plot = gg_nn_fit, filename = "nn-fit")

# also create a smaller dataset for fitting GPs
smaller_ratio <- 4
ml_pred_small_df <- filter(ml_pred_df, seq(ml_pred_n) %% smaller_ratio == 0)
```

04-basis-functions.R

```
# basis-functions ----

# create a list of basis functions
basis_functions <- list(
  "x" = ~.,
  "x_squared" = ~ .^2,
  "sin_x" = ~ sin(.),
  "sin_2x" = ~ sin(2 * .),
  "sin_half_x" = ~ sin(. / 2),
  "cos_x" = ~ cos(.),
  "cos_2x" = ~ cos(2 * .),
  "cos_half_x" = ~ cos(. / 2)
)

# create a lookup table to help export tables to LaTeX using knitr::kable()
basis_functions_latex <- tibble(
  name = c("x", "x_squared", "sin_x", "sin_2x", "sin_half_x",
           "cos_x", "cos_2x", "cos_half_x"),
  latex = c("x", "x^2", "\\sin(x)", "\\sin(2x)", "\\sin(x/2)",
            "\\cos(x)", "\\cos(2x)", "\\cos(x/2)")
) %>%
  mutate(
    latex_old = math_mode(latex, "$"),
    latex = math_mode(latex)
  ) %>%
  bind_rows(c(name = "(Intercept)", latex = "Intercept", latex_old = "Intercept"), .)

# create a dataframe of the pedictors for regression
mr_train_df <- map_df(basis_functions, ~ map_dbl(ml_pred_x, .)) %>%
  mutate(y = ml_pred_y)

# also create a smaller dataframe of the pedictors for use with GPs
mr_train_small_df <- filter(mr_train_df, seq(ml_pred_n) %% smaller_ratio == 0)
```

05-multiple-regression.R

```
# multiple regression ----
```

```r
# fit a linear model
mr_model <- lm(y ~ . + 1, data = mr_train_df)

# output the results of the linear model
mr_model %>%
  get_model_summary() %>%
  save_table(
    label = "multiple-regression",
    caption = "The results of the multiple regression model."
  )

# predict from the multiple regression model
mr_pred_df <- predict(mr_model, mr_train_df) %>%
  as.numeric() %>%
  tibble(
    x = mr_train_df$x,
    y = .,
    model = factor("mr", levels = model_types)
  )

# plot the fit
gg_mr_fit <- bind_rows(mr_pred_df, ml_pred_df) %>%
  ggplot(mapping = aes(x = x, y = y, colour = model)) +
  geom_line() +
  scale_colour_manual(values = model_colours) +
  cartesian_labels()
save_plot(plot = gg_mr_fit, filename = "multiple-regression-fit")
```

06-stepwise-regression.R

```r
# stepwise regression ----

# initial model
sr_model_empty <- lm(y ~ 0, data = mr_train_df)

# reduce the model using stepwise regression
sr_model <- stepAIC(sr_model_empty,
  scope = list(upper = mr_model, lower = sr_model_empty),
  direction = "both",
  trace = FALSE
)

# output the results of the reduced model
sr_model %>%
  get_model_summary() %>%
  save_table(
    label = "stepwise-regression",
    caption = "The results of the model reduced using stepwise \\ac{AIC}."
  )

# predict from the stepwise regression model
sr_pred_df <- predict(sr_model, mr_train_df) %>%
  as.numeric() %>%
  tibble(
    x = mr_train_df$x,
```

```
    y = .,
    model = factor("sr", levels = model_types)
  )

# plot the fit
gg_sr_fit <- bind_rows(sr_pred_df, ml_pred_df) %>%
  ggplot(mapping = aes(x = x, y = y, colour = model)) +
  geom_line() +
  scale_colour_manual(values = model_colours) +
  cartesian_labels()
save_plot(plot = gg_sr_fit, filename = "stepwise-regression-fit")
```

07-lasso.R

```
# lasso ----

# reduce the model using lasso regularisation
lasso_model <- mr_train_df %>%
  select(-y) %>%
  as.matrix() %>%
  glmnet(x = .,
         y = mr_train_df$y,
         family = "gaussian",
         lambda = exp(seq(2, -3, len = 100)))

# perform leave-one-out cross validation to optimise lambda
lasso_cv <- cv.glmnet(
  x = as.matrix(select(mr_train_df, -y)),
  y = mr_train_df$y,
  lambda = exp(seq(2, -3, len = 50)),
  nfolds = ml_pred_n,
  grouped = FALSE
)

# plot changing lambda over time
gg_lasso_lambda <- lasso_model$beta %>%
  as.matrix() %>%
  t() %>%
  as_tibble() %>%
  mutate(`(Intercept)` = as.numeric(lasso_model$a0)) %>%
  mutate_all(~ replace(., abs(.) < 1e-4, NA)) %>% # remove when estimate is zero
  mutate(log_lambda = log(lasso_model$lambda)) %>%
  gather(key = "term", value = "estimate", -log_lambda, na.rm = TRUE) %>%
  inner_join(basis_functions_latex, by = c("term" = "name")) %>%
  mutate(Term = latex_old) %>%
  arrange(match(Term, basis_functions_latex$latex_old)) %>% # sort terms
  ggplot(mapping = aes(x = log_lambda, y = estimate, colour = Term)) +
  geom_vline(xintercept = log(lasso_cv$lambda.1se),
             linetype = "dashed",
             colour = "red",
             alpha = 0.5) +
  geom_line() +
  scale_y_continuous(limits = c(NA, 5)) +
  scale_colour_manual(values = unnamed_colours) +
  labs(x = math_mode("\\log(\\lambda)", "$")) +
```

```
  theme(legend.position = "bottom")
save_plot(plot = gg_lasso_lambda,
          filename = "lasso-lambda",
          aspect_ratio = 0.8 * golden)

# extract the optimised coefficients
lasso_coefs <- lasso_cv %>%
  coef(s = "lambda.1se") %>%
  as.matrix() %>%
  as_tibble(rownames = "term") %>%
  rename(estimate = `1`) %>%
  filter(abs(estimate) > 1e-4) %>%
  inner_join(basis_functions_latex, by = c("term" = "name")) %>%
  select(term = latex, estimate) %>%
  set_names(str_to_sentence(names(.)))

# save the coefficients as a table
save_table(lasso_coefs,
  label = "lasso-coefs",
  caption = "The optimised coefficients using leave-one-out cross-validation on the value of
)

# predict from the lasso model
lasso_pred_y <- mr_train_df %>%
  select(-y) %>%
  as.matrix() %>%
  predict(lasso_cv, newx = ., s = "lambda.1se") %>%
  as.numeric()

lasso_pred_df <- tibble(
  x = mr_train_df$x,
  y = lasso_pred_y,
  model = factor("lasso", levels = model_types)
)

# plot the fit
gg_lasso_fit <- bind_rows(lasso_pred_df, ml_pred_df) %>%
  ggplot(mapping = aes(x = x, y = y, colour = model)) +
  geom_line() +
  scale_colour_manual(values = model_colours) +
  cartesian_labels()
save_plot(plot = gg_lasso_fit, filename = "lasso-fit")
```

08-compare-coefs.R

```
# compare-coefs ----

# compare the coefficients of the regression models
bind_rows(
  mutate(get_model_summary(mr_model), model = "mr"),
  mutate(get_model_summary(sr_model), model = "sr"),
  mutate(lasso_coefs, model = "lasso")
) %>%
  select(Term, Estimate, model) %>%
  spread(key = model, value = Estimate) %>%
```

```
  arrange(match(Term, basis_functions_latex$latex)) %>%
  select(Term, mr, sr, lasso) %>%
  rename(
    "Term" = Term,
    "Multiple\nregression" = mr,
    "Stepwise\nregression" = sr,
    "LASSO" = lasso
  ) %>%
  set_names(linebreak(names(.), align = "r")) %>%
  save_table(
    label = "compare-coefs",
    caption = "The coefficients of the three linear regression models."
  )
```

09-gaussian-process.R

```
# gaussian process ----

# fit a gaussian process model
gp_model <- mlegp(X = ml_pred_small_df$x, Z = ml_pred_small_df$y)

# predict from the GP inclusing standard errors
confidence_level <- 0.99
gp_pred_df <- predict(gp_model, newData = matrix(ml_pred_x), se.fit = TRUE) %>%
  map(as.numeric) %>%
  as_tibble() %>%
  rename(y = fit, se = se.fit) %>%
  mutate(
    x = ml_pred_x,
    y_lwr = y - qnorm((1 - confidence_level) / 2) * se,
    y_upr = y + qnorm((1 + confidence_level) / 2) * se,
    model = factor("gp", levels = model_types)
  )

gg_gp_fit <- gp_pred_df %>%
  bind_rows(ml_pred_df) %>%
  ggplot(mapping = aes(x, y, ymin = y_lwr, ymax = y_upr, colour = model, fill = model)) +
  geom_ribbon(linetype = 0,alpha=0.5) +
  geom_line() +
  scale_colour_manual(values = model_colours) +
  cartesian_labels()
save_plot(plot = gg_gp_fit, filename = "gp-fit")
```

10-regression-and-gp.R

```
# regression and gp ----

# predict from lasso model
lasso_pred_small_y <- mr_train_small_df %>%
  select(-y) %>%
  as.matrix() %>%
  predict(lasso_cv, newx = ., s = "lambda.1se")

# calculate residuals
lasso_resid <- ml_pred_small_df$y - lasso_pred_small_y
```

```r
lasso_resid_df <- tibble(
  x = ml_pred_small_df$x,
  y = lasso_resid,
  model = "residual"
)

# fit a gaussian process model
gp_resid_model <- mlegp(X = mr_train_small_df$x, Z = lasso_resid)

# predict from the GP
gp_resid_pred <- predict(gp_resid_model, newData = matrix(mr_train_small_df$x)) %>%
  as.numeric()

combined_pred_df <- tibble(
  x = ml_pred_small_df$x,
  lasso = lasso_pred_small_y,
  gp = gp_resid_pred,
  y = lasso + gp,
  model = factor("combined", levels = model_types)
)

# plot the GP fitting the residuals
gg_gp_resids_fit <- ggplot(data = combined_pred_df, mapping = aes(x, y = gp)) +
  geom_point() +
  geom_line() +
  scale_colour_manual(values = model_colours) +
  labs(x = "$x$", y = "residuals")
save_plot(plot = gg_gp_resids_fit, filename = "gp-resids-fit")

# plot the fit
gg_combined_fit <- bind_rows(combined_pred_df, ml_pred_df) %>%
  ggplot(mapping = aes(x, y, colour = model)) +
  geom_line() +
  scale_colour_manual(values = model_colours) +
  cartesian_labels()
save_plot(plot = gg_combined_fit, filename = "combined-fit")
```

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.

Dancik, G. M. and Dorman, K. S. (2008). 'mlegp: statistical analysis for computer models of biological systems using R'. *Bioinformatics* 24.17, p. 1966.

Friedman, J., Hastie, T. and Tibshirani, R. (2010). 'Regularization Paths for Generalized Linear Models via Coordinate Descent'. *Journal of Statistical Software* 33.1, pp. 1–22. URL: `http://www.jstatsoft.org/v33/i01/`.

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.

R Core Team (2018). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria. URL: https://www.R-project.org/.

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.

Smit, J. (2018). 'Can statistics help us to understand deep learning? Literature Review'.

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.