Functional Neural Networks for Scalar Prediction

by

Barinder Thind

B.Sc. Honors, Simon Fraser University, 2018

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
Department of Statistics & Actuarial Science
Faculty of Science

SIMON FRASER UNIVERSITY
Spring 2020

Approval

Name:	**Barinder Thind**
Degree:	**Master of Science (Statistics)**
Title:	**Functional Neural Networks for Scalar Prediction**
Examining Committee:	**Chair:** David Stenning Assistant Professor Department of Statistics & Actuarial Science Simon Fraser University
	Jiguo Cao Supervisor Associate Professor Department of Statistics & Actuarial Science Simon Fraser University
	Liangliang Wang Committee Member Associate Professor Department of Statistics & Actuarial Science Simon Fraser University
	Lloyd Elliott Examiner Assistant Professor Department of Statistics & Actuarial Science Simon Fraser University
Date Defended:	**April 7, 2020**

Abstract

We introduce a methodology for integrating functional data into densely connected feed-forward neural networks. The model is defined for scalar responses with at least one functional covariate and some number of scalar covariates. A by-product of the method is a set of functional parameters that are dynamic to the learning process which leads to interpretability. The model is shown to perform well in a number of contexts including prediction of new data and recovery of the true underlying coefficient function; these results were confirmed through cross-validations and simulation studies. A collection of useful functions are built on top of the `Keras/Tensorflow` architecture allowing for general use of the approach.

Keywords: Functional Data Analysis, Machine Learning, Neural Networks, Prediction

Dedication

I would like to dedicate this work to my grandma and to my family pet.

They both unfortunately, passed away while I was completing this degree.

Acknowledgements

I wanted to take this space to appreciate the relationships I have with the people in my life. In particular, I want to thank my supervisor Jiguo Cao, for his encouraging words and meaningful contributions to the research done here. To everyone in my cohort for being an awesome group of people. To my closest friends and many others, thank you for being there. I'm grateful that professors like Richard, Rachel, Jinko, Ian, Tom, Harsha, Steve, Carl, Robin, Tim, Joan, Brad, and Derek provided me with wisdom that was, and still is, far beyond my years. I am appreciative of Charlene, Sadika, and Jay for making my life so much easier. I am so thankful to Marie for all that she did for me; I truly looked forward to our spontaneous conversations! A thank you as well to Lloyd, Liangliang, and David for providing me valuable feedback on my thesis and agreeing to be on the examining committee. Also, a thanks to Dr. Ng. And finally, I am so grateful for my family. I am glad to have grown closer to my brother over these past two years and it goes without saying, my parents have provided me with more than I could ever put into words.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The ever-expanding umbrella that encompasses deep-learning methodologies has been limited to the multivariate scope, thus excluding usage of functional covariates. With the advent and rise of *functional data analysis* (FDA) (Ramsay and Silverman, 2010), it is natural to extend neural networks to this space. The main goal of this article is to provide a new means of modelling functional data for scalar response prediction. Previous methods include the classical *functional linear model* defined as (Cardot et al., 1999):

$$E(Y|X) = \alpha + \int_{\mathcal{S}} \beta(s)X(s)\,\mathrm{d}s,$$

where Y is the scalar response, $X(s)$ is the functional covariate, α is the intercept term, $\mathcal{S}$ is the domain over which the functional data is observed, and $\beta(s)$ is the functional coefficient. Note, it is useful to represent $\beta(s)$ in a functional form $\beta(s) = \sum_j c_j\phi_j(s)$, where $\{\phi_j(s)\}$ forms a basis (Cardot et al., 2003). In this scheme, in order to estimate $\beta(s)$, the error is minimized: $\beta^*(s) = \min_{\beta(s),\alpha} \left\{\sum_i \left(y_i - \alpha - \int_{\mathcal{S}} \beta(s)x_i(s)\,\mathrm{d}s\right)^2\right\}$ (Hastie and Mallows, 1993). However, this can result in volatile estimates along with some significant probability of over-fitting (Ramsay and Silverman, 2010). Instead, a constrained approach can be used where we penalize the second derivative. This approach was extended to the general linear model case and took the following form:

$$E(Y|X) = g\left(\alpha + \int_{\mathcal{S}} \beta(s)X(s)\,\mathrm{d}s\right)$$

where $g(\cdot)$ is referred to as the *link* function (Müller et al., 2005). When this function $g(\cdot)$ has no parametric form, the model is referred to as the *functional single index model* (Jiang et al., 2011). Other predictive methods in the realm of FDA are related to the estimation of $\beta(s)$. For example, a partial least squares approach was used by Preda et al. (2007) where an attempt was made to maximize the covariance between the response, Y and the functional covariate, $X(s)$. We can also consider a non-parametric approach as seen in Ferraty and Vieu (2006) where the model is:

$$E(Y|X) = r\left(X(s)\right).$$

In this case, $r(\cdot)$ is a smooth function that is estimated using a kernel approach. Another method, which serves as an extension to the previous, is that of the *semi-functional partial linear model* (Aneiros-Pérez and Vieu (2006)) defined as:

$$E(Y_i|X_i) = r\left(X_i(s)\right) + \sum_p \beta_{ip} Z_{ip},$$

where Z is the non-functional covariate that is observed in the usual multivariate case. Again, the function $r(\cdot)$ is estimated using kernel methods. All of these approaches have shown to have some level of predictive success. However, we show that the general neural network proposed here outperforms these approaches.

To the best of our knowledge, this is the first neural network approach that details the methodology for when the initial inputs are functions; we provide the generalization for J functional covariates and K scalar covariates along with a way to estimate the true underlying coefficient function, $\beta(s)$. The form of a single neuron v_i, in this model is written as:

$$v_i = g\left(\sum_{k=1}^{K} \int_{\mathcal{S}} \beta_k(s) x_{ik}(s)\,\mathrm{d}s + \sum_{j=1}^{J} w_j^{(1)} z_{ij} + b^{(1)}\right) \tag{1.1}$$

where $g(\cdot)$ is some activation function, $\boldsymbol{w}$ is the vector of weights associated with the scalar covariates, the superscript represents the layer number, and $\beta_k(s)$ is the set of *functional neural coefficients* (FNCs) that weigh the functional covariates.

We are motivated to pursue this marriage of FDA and neural networks because neural networks have shown to supersede previous benchmarks, and functional analyses serve as powerful inference techniques. Recent advances in deep learning have been empirically shown to outperform other methods in a number of important problems. For example, the (Krizhevsky et al., 2012) approach won the *ImageNet Large-Scale Visual Recognition Challenge* in 2012 beating out the next best approach by an error rate that was over 10% less. In 2015, (He et al., 2016) introduced *ResNets* which allowed for circumvention of vanishing gradients – an innovation that carved a path for networks with exponentially more layers thus further improving error rates. These successes however, have come at the cost of interpretability. As the models become more complex, it becomes an increasingly difficult task to make sense of the network parameters. Classical linear regression models have a relatively clear interpretation of the parameters estimates (Seber and Lee, 2012). In the functional linear regression case, the coefficient parameters being estimated are functions $\boldsymbol{\beta(s)}$, rather than a set of scalar values, $\boldsymbol{\beta}$. This paper details an approach that makes the functional coefficient traditionally found in the regression model readily available from the neural network process as the FNCs alluded to earlier; the expectation is that this allows for interpretation from a set of models with superior predictive power.

Our paper contributes three main components: first, we introduce the methodology for *functional neural networks* which can be found in Chapter 2. Additionally, commentary is provided on the inferential potential, weight initialization, and the hyperparameters of these networks. Then, Chapter 3 provides results from real world examples; this includes prediction comparisons between a number of methods for multiple data sets. The first example concerns the relationship between daily bike rentals and temperature curves. The second and third are applications of this approach to predict meat contents using absorbance curves and predict total precipitation using temperature curves, respectively. In Chapter 4, we use simulation studies for the purpose of recovering the true underlying coefficient

function $\beta(s)$, and to test the predictive accuracy of multivariate and functional methods in four different contexts. Lastly, Chapter 5 contains some closing thoughts and new avenues of research for this kind of network.

Chapter 2

Methodology

2.1 Functional Neural Networks

The usual neural network is made up of *hidden layers* each of which contains some number of *neurons*. We can index these as n_1 to n_u where the subscript u refers to the u^{th} hidden layer and n_u is the number of neurons in that particular layer. Each neuron (in each layer) is some (potentially) non-linear transformation of a linear combination of each activation in the previous layer. An *activation* value is the output from each of these neurons. For example, the first hidden layer $\boldsymbol{v}^{(1)}$ would be defined as

$$\boldsymbol{v}^{(1)} = g\left(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right),$$

where $\boldsymbol{W}^{(1)}$ and $\boldsymbol{b}^{(1)}$ are the weights and biases and $g(\cdot)$ is some *activation function* that transforms the resulting linear combination (Tibshirani et al., 2009). In this example, the dimensionality of $\boldsymbol{W}^{(1)}$ would be (n_1 x J) where J is the number of scalar covariates associated with each observation, i. The choice of the function $g : \mathbb{R}^{n_1} \to \mathbb{R}^{n_1}$ is highly context dependent. The rectifier (Glorot et al., 2011) and sigmoidal functions (Han and Moraga, 1995) have been shown to have useful properties which makes them useful choices for g. Note that the vector $\boldsymbol{x}$ corresponds to a single observation of our data set. The resulting vector $\boldsymbol{v}^{(1)}$ is n_1-dimensional. This vector contains the activation *values* to be passed on to the next layer.

Thus far, the assumption has been that $\boldsymbol{x}$ is J dimensional. However, we wish now to consider the case when our input is infinite dimensional defined over some domain $\mathcal{S}$, i.e.,

we postulate our input is a function $x(s) : \mathcal{S} \to \mathbb{R}$, $s \in \mathcal{S}$. We must weigh this covariate at every point along its domain and so, as in the case with the functional linear model, our coefficient must be infinite dimensional as well. We define this coefficient as $\beta(s)$. The form of a neuron with a single functional covariate then becomes

$$v_n^{(1)} = g\left(\int_{\mathcal{S}} \beta_n(s)x(s)\,\mathrm{d}s + b_n^{(1)}\right), \tag{2.1}$$

where the subscript n is an index that denotes one of the n_1 neurons in this first hidden layer, i.e., $n \in \{1, 2, \ldots, n_1\}$. We omit the superscript on the functional parameter $\beta(s)$, because this parameter only exists in the first layer of the network. The functional covariate $x(s)$, is passed into the network as its basis expansion. Since our estimations will likely not be exact interpolations of the discrete data points, the assumption is made that the difference between the functional fit and the discrete values is a matter of random error. That is, $y = x(s) + \epsilon$, where $x(s) = \sum_{p=1}^{P} c_p\phi_p(s) = \boldsymbol{c}^{\,T}\boldsymbol{\phi}(s)$. The coefficients $\boldsymbol{c}$ can be estimated with some minimization approach, e.g., $\hat{c} = (\Phi^T\Phi)^{-1}\Phi^T y$ where Φ is the n_1 by P matrix containing the evaluations of the P basis functions at some pre-specified values of s. We remind the reader that while we accrue some error in the smoothing process, we also reduce the stochastic error associated with the raw discrete data; this trade-off can circumvent the overfitting nature of universal approximators such as neural networks.

The same treatment is given to $\beta(s)$ in that it is written as a linear combination of basis functions. The coefficients on the basis expansion for $\beta(s)$ will be initialized by the network; these initializations will then be updated as the network *learns*. The choice of basis is a hyperparameter in both cases with *b-splines*, polynomial expansions, and *Fourier* functions being common choices. We also note that the evaluation of the neuron in (2.1) results in some scalar value - this implies that the rest of the $U-1$ layers of the network can be of any of the usual forms (feed-forward, convolution, recurrent, etc.). Using these basis approximations of $x(s)$ and $b(s)$, we can apply a special case of Fubini's theorem which states that for general f_n, if $\int \sum |f_n| < \infty$ or $\sum \int |f_n| < \infty$, then $\int \sum f_n = \sum \int f_n$ (Fubini,

1907). The form of a single neuron then becomes:

$$v_n^{(1)} = g\left(\int_{\mathcal{S}} \beta_n(s)x(s)\,\mathrm{d}s + b_n^{(1)}\right) \tag{2.2}$$

$$= g\left(\int_{\mathcal{S}} \sum_{m=1}^{M} c_{nm}\phi_{nm}(s)x(s)\,\mathrm{d}s + b_n^{(1)}\right) \tag{2.3}$$

$$= g\left(\sum_{m=1}^{M} c_{nm}\int_{\mathcal{S}} \phi_{nm}(s)x(s)\,\mathrm{d}s + b_n^{(1)}\right), \tag{2.4}$$

where the integral in (2.4) can be approximated if need be, such as in the case when the basis expansion of $\beta(s)$ or $x(s)$ is done using B-splines.

We can now consider the generalization for K functional covariates and J scalar covariates. Consider the input layer as presented in Figure 2.1. The values correspond to the i^{th} functional observation and can be seen as the set:

$$\text{input} = \{x_1(s), x_2(s), ..., x_K(s), z_1, z_2, ..., z_J\}.$$

Then the n^{th} neuron of the first hidden layer corresponding to the i^{th} functional observation can be formulated as:

$$v_n^{(1)} = g\left(\sum_{k=1}^{K}\int_{\mathcal{S}} \beta_{kn}(s)x_k(s)\,\mathrm{d}s + \sum_{j=1}^{J} w_{jn}^{(1)}z_j + b_n^{(1)}\right),$$

where

$$\beta_{kn}(s) = \sum_{m=1}^{M} c_{knm}\phi_{knm}(s)$$

and $\phi_m(s)$ is as usual, the evaluation of the m^{th} basis function at some value s. This neuron formulation is the core of this methodology as alluded to in (1.1). Note that c_m here is unique at the initialization for each weight function, $\beta(s)$. The choice of these initializations is discussed later in the article. We also note that in this formation, we have assumed that M is the same across all K functional weights. However, this is for the sake of brevity and the the choice of the hyperparameter for each of the K functional covariates is left at the

discretion of the user. It could be the case that the user prefers some weight functions to be defined using a different number of basis terms than M, say M_k.

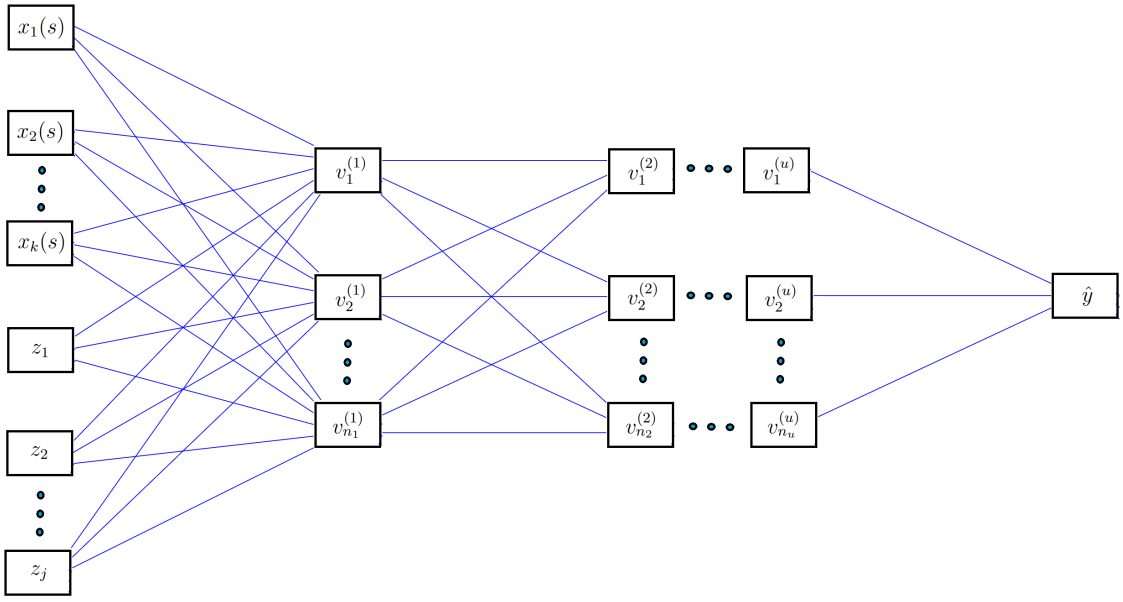

Figure 2.1: The form of the general functional neural network for when the inputs are a combination of functions, $x_k(s)$, and scalar values, z_j. The response/output of this network is a scalar value, $\hat{y}$.

Having specified the form, we define the following formation of the first layer in general:

$$\begin{aligned} v_n^{(1)} &= g\left(\sum_{k=1}^{K}\int_{\mathcal{S}}\sum_{m=1}^{M_k} c_{kmn}\phi_{kmn}(s)x_k(s)\,\mathrm{d}s + \sum_{j=1}^{J} w_{jn}^{(1)} z_j + b_n^{(1)}\right) \\ &= g\left(\sum_{k=1}^{K}\sum_{m=1}^{M_k} c_{kmn}\int_{\mathcal{S}}\phi_{kmn}(s)x_k(s)\,\mathrm{d}s + \sum_{j=1}^{J} w_{jn}^{(1)} z_j + b_n^{(1)}\right). \end{aligned}$$

To justify this, we consider the one-layer case and use `Theorem 1` and `Lemma 1` as defined in (Cybenko, 1989). The proof states that linear combinations of the form $G(x) = \sum_{n=1}^{N}\alpha_n\sigma\left(y_n^T z + \theta_n\right)$ exhibit the quality that under some conditions, the function you want to learn, f, differs from $G(x)$ by only some error $\epsilon > 0$. Since we are looking at the one-layer case, we have n_1 neurons (indexed from $n = 1$ to n_1) and we omit the superscript that indexes the layer number, $u = 1$. Additionally, we fix the observation number i since it does not play a role in the proof (you can apply the same argument to each observation).

Theorem 1. *Let $g : \mathbb{R} \to \mathbb{R}$ be any continuous sigmoidal function, I_n denote the $n-$dimensional hypercube $[0,1]^n$ and $\mathcal{C}(I_n)$ denote the space of continuous functions. Then, the finite sum of the following form, is dense in $\mathcal{C}(I_n)$:*

$$h(s) = \sum_{n=1}^{n_1} \Psi_n g \left(\sum_{k=1}^{K} \left(\int_{\mathcal{S}} \beta_{nk}(s) x_k(s) \, \mathrm{d}s \right) + \sum_{j=1}^{J} w_{nj} z_j + b_n \right),$$

meaning that for any $f(s) \in \mathcal{C}(I_n)$ and for $\epsilon > 0$, the function $h(s)$ obeys:

$$|h(s) - f(s)| < \epsilon$$

A proof is provided in the Appendix. After running through this set of initial neurons and calculating the activations for the layers following, we can arrive at a final value. The output will be single dimensional, $\hat{y}$. In order to assess performance, we can use some loss function, R; for example, *mean squared error*

$$R(\theta) = \sum_{i=1}^{N} \left(y_i - \hat{y}_i(\theta) \right)^2,$$

where θ is the set of parameters defining the network and y_i is the true scalar response.

2.2 Network Training

Having defined the general formation of *functional neural networks* (FNNs), we can now turn our attention to the optimization of this kind of network. We will consider the usual backpropogation algorithm (Rumelhart et al., 1985). While in the implementation, we used an `adam()` optimizer (Kingma and Ba, 2014), we can explain the general process when the optimization scheme uses stochastic gradient descent.

Given our generalization and reworking of the parameters in the network, we can note that the set $\theta^{'}$ making up the gradient associated with the parameters is:

$$\theta^{'} = \left\{ \bigcup_{k=1}^{K} \bigcup_{m=1}^{M_k} \bigcup_{n=1}^{n_1} \frac{\partial R}{\partial c_{kmn}}, \bigcup_{u=1}^{U} \bigcup_{j=1}^{J_u} \bigcup_{n=1}^{n_u} \frac{\partial R}{\partial w_{ujn}}, \bigcup_{u=1}^{U} \bigcup_{n=1}^{n_u} \frac{\partial R}{\partial b_{un}} \right\},$$

This set exists for every observation, i. We are trying to optimize for the entirety of the training set, so we will move slowly in the direction of the gradient. The rate at which we move (the *learning rate*) will be denoted by γ. For the sake of efficiency, we will take a subset of the training observations (a *mini batch*) for which we calculate θ'_{sub}, where the subscript *sub* refers to the fact that it is a mini batch. Then, letting $\bar{a} = \sum_i^N \frac{a'_i}{N}$, where $a'_i = \dfrac{\partial R}{\partial a_i}$ is the derivative of any parameter $a_i \in \theta$ for the i^{th} observation and N is the size of the mini batch, we observe that the update for a is $a = a - \gamma\bar{a}$. We summarize the entire network process in Algorithm 1.

Lastly, we would like to emphasize that the number of parameters in the network presented here has decreased significantly under this approach. Consider a longitudinal data set where we have n observations and J_{ml} scalar repeat measurements of some covariate at different points along a continuum. Passing this information into a network will mean that the number of parameters in the first layer will be $(J_{ml} + 1) \cdot n_1$. Note that when we define a functional observation, it is good practice that the number of basis functions M is less than the number of observed points, J_{ml} to avoid overfitting via interpolation. Therefore, the number of parameters in the first layer of our network is $(M + 1) \cdot n_1$ where $M < J_{ml}$.

2.3 Functional Neural Coefficients

Since a leading contributor to the black-box reputation of neural networks is the inordinate amount of changing weights and biases, it would be helpful to consider rather a function defined by these seemingly uninterpretable numbers. In a *functional neural network*, one set of weights we are estimating defines a basis function (e.g., $\beta_k(s) = \sum_{m=1}^{M} c_{km}\phi_{km}(s)$). These basis functions are akin to the ones predicted in the functional regression model (Ramsay and Silverman, 2010); the final set of weights here define the M-basis β coefficient weight functions which can be compared with the one estimated from a function linear model. This extracted function will be referred to as the *functional neural coefficient*. In the case of multiple neurons, we take the average of the estimated coefficients; that is, if we are considering c_m, then the estimated value will be $c_m = \frac{\sum_{n=1}^{n_1} c_{nm}}{n_1}$. Over iterations of the

Algorithm 1: Functional Neural Networks

Input: Functional and Scalar Observations
Output: θ

---

1. Set Hyperparameters:
   γ, # of Basis Functions, Activation Functions, # of Layers, # Of Neurons per Layer, Epochs, Loss Function
2. Initialize weights of network, θ_p
3. **for** j in 1:*Epochs*
   - 3i. Forward Pass
     - a. Observed data passed to first hidden layer
     - b. Approximate $\int_{\mathcal{S}} \phi_{km}(s)x(s)\,\mathrm{d}s = \tilde{\phi}_{km}$ for each basis function, m and for each functional covariate, k
     - c. Calculate $g\left(\sum_{k=1}^{K}\sum_{m=1}^{M_k} c_{km}\tilde{\phi}_{km} + \sum_{j=1}^{J} w_j^{(1)} z_j + b^{(1)}\right)$ for each neuron
     - d. Pass activations in c. to any other network architecture as per usual
     - e. Calculate loss: $R(\theta)$
   - 3ii. Backward Pass
     - a. Compute θ'
     - b. $\forall\ a \in \theta_p$, update a as: $a = a - \gamma\bar{a}$
   - 3iii. **If** $j \leq$ *Epochs*
     - a. Go to 3i.
     - b. **Else:** Go to 4.
4. Return $\theta = \theta_p$

---

network, as it is trained, we can see movement of the functional coefficient. Figure 2.2 may be illuminating.

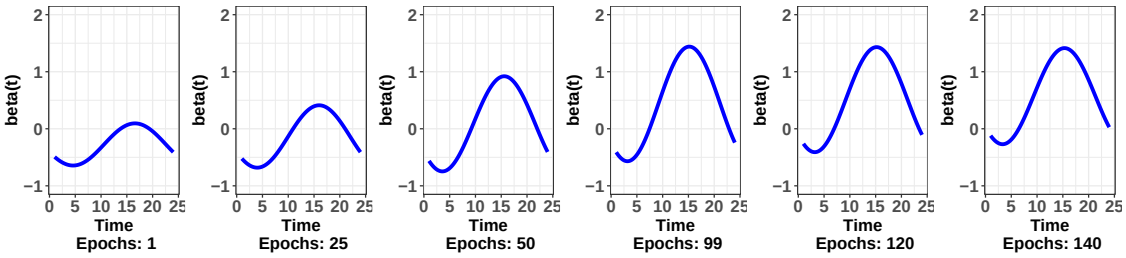

Figure 2.2: An example of how the FNC changes over varying the number of epochs. At 99, the validation error stops decreasing (with respect to some threshold). We can see that the difference in the curves is most pronounced in the beginning and is least pronounced after the model finds some local extrema. In this example, the weights were initialized from a uniform distribution.

2.4 Weight Initialization and Parameter Tuning

For any usual neural network, the weights and biases can be initialized in a number of ways. For example, in (Kim and Ra, 1991) weights are initialized based on a boundary that allowed for faster convergence. Another approach is to consider a zero-initialization i.e., letting the initial parameter values be 0. Many of these approaches have also been compared to one another using various guidelines (Fernández-Redondo and Hernández-Espinosa, 2001). In the case of the networks presented here, this is left as a hyperparameter. Since the implementation is built on top of `Keras`, the initialization is dependant on the type of connected layer, but generally the `glorot_uniform()` initializer (which is a unifrom distribution that has parameters depending on the dimensionality of the inputs) is the choice (for *dense* layers) (Chollet et al., 2015).

Due to the sheer number of hyperparameters in the network, a tuning function is provided. The function `FNN_Tune()` will take in a list of possible values for each parameter and run a cross-validation for all combinations. The number of folds is left at the discretion of the user. The general scheme is that the function creates a grid, calculates the cross-validated mean squared prediction error

$$\text{MSPE}_{CV} = \frac{\sum_i \left(x_i^{\text{CV}} - x_i^{\text{true}}\right)^2}{n},$$

where the dimensionality of the CV set depends on the number of folds, and outputs the combination with the minimum value of this criterion. A complete list of hyperparameters is given in the Appendix.

One important parameter in this particular kind of network is the number of basis functions that govern the functional weights. Tuning this is fairly important as the number of terms significantly impacts the potential for interpretability and restricts us to some particular shape of the curve. In the examples to come, we tune for this hyperparameter using the tuning function mentioned above.

Chapter 3

Real World Verifications

3.1 Bike Rental Data

An important problem in rental businesses is the amount of supply to keep on-site. If the company cannot meet demands, they are missing out on profit, and if they exceed the required supply, they have made investments that are not yielding an acceptable return. Using the bike rental data set (Fanaee-T and Gama, 2014), we look to model the relationship between the total number of daily rentals and hourly temperature throughout the day. The 102 functional observations are made using a 31-basis fourier expansion for each Saturday of a week.

We are first concerned with the accuracy of our predictions. Using R^2 and a 10-fold cross-validated MSPE, we can compare results for a number of models. Here, we compare with the usual functional linear model, an fPCA approach, a non-parametric functional linear model, and a functional partial least squares model. The results are summarized in Table 3.1. We observe that FNNs outperform all the other models using both criteria but note that the penalized partial least squares approach and the principal ridge regression performed comparably.

We can also look to see what the determined relationship is according to these models between temperature and daily rentals as indicated by $\beta(s)$. In Figure 3.1, the estimated functions are given. For the functional linear model, we note that there seems to be no obvious discernable relationship between temperature and bike rentals. In the case of the FNC, we see that there seems to be a positive relationship as we move into the afternoon

and that this relationship tapers off as the day ends. We would also expect there to be no effect for when bike rental retailers would be closed, and this is much better reflected in the FNC than the linear model coefficient curve.

Model	$\texttt{MSPE}_{CV}$	R^2
Functional Linear Model (Basis)	0.0723	0.515
Functional Non-Parametric Regression	0.143	0.154
Functional PC Regression	0.0773	0.503
Functional PC Regression (2nd Deriv Penalization)	0.128	0.0481
Functional PC Regression (Ridge Regression)	0.0823	0.464
Functional Partial Least Squares	0.0755	0.458
Functional Partial Least Squares (2nd Deriv Penalization)	0.0701	0.545
Functional Neural Networks	**0.0669**	**0.582**

Table 3.1: Tabulated 10 fold cross-validated mean-squared predication error and R^2 of eight models, including the FNN. The italics indicate second-best model performance, whereas the bolded (and green coloured) cells indicate best performance.

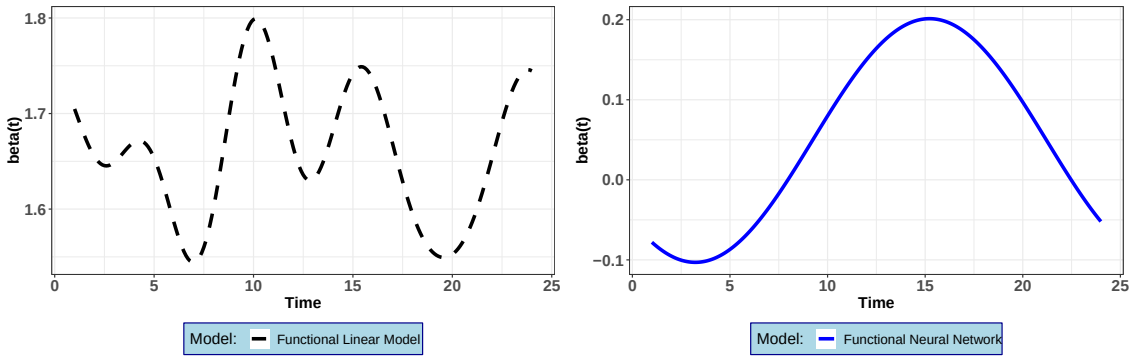

Figure 3.1: The estimated coefficient functions for the usual functional linear model and the functional neural network for the bike rental data set. The plot on the left is a functional linear model and the plot on the right is the FNN. The optimal number of basis functions was 11 for the linear model and 3 for the neural network.

3.2 Tecator Data

We consider the classic *Tecator* data set (Thodberg, 2015). The data are recorded on a Tecator Infratec Food and Feed Analyzer working in near-infrared wavelength range:

850 nm - 1050 nm. Each sample contains meat with different moisture, fat, and protein contents. The goal is to predict the fat contents of some given meat sample using the functional covariate of the near infrared absorbance spectrum and the scalar covariate associated with the water contents. Absorbance spectroscopy measures the fraction of incident radiation absorbed by the sample. Samples with higher water composition may exhibit different spectral features (absorbance bands) than samples with higher protein content. So using functional information, we expect to perform better than other methods because we can provide information about the derivatives to the network – embedding more of the associated physics into the learning task.

Model	MEP	R^2
fregre.basis(X.d1, Fat)	0.0626	0.928
fregre.basis.cv(X.d2, Fat)	0.0566	0.965
fregre.pc(X.d1, Fat)	0.0580	0.950
fregre.pc(X.d2, Fat)	0.0556	0.954
fregre.pls(X.d1, Fat)	0.0567	0.951
fregre.pls(X.d2, Fat)	0.0487	0.962
fregre.lm(Fat, X.d1 + Water)	0.0097	0.987
fregre.lm(Fat, X.d2 + Water)	0.0119	0.986
fregre.np(X.d1, Fat)	0.0220	0.987
fregre.np(X.d2, Fat)	0.0144	0.996
fregre.plm(Fat, X.d1 + Water)	0.0090	0.996
fregre.plm(Fat, X.d2 + Water)	0.0115	**0.997**
FNN(Fat, X.d2 + Water)	**0.00883**	0.965

Table 3.2: This table compares the results of our method to those used in (Bande and Fuente, 2012). We can observe that our approach was nearly the best with respect to MEP. The italics indicate second-best model performance, whereas the bolded (and green coloured) cells indicate best performance. Note that we only presented the results using the second derivative as the covariate since it was the better performer.

In total, there are 215 functional observations. The first 165 absorbance curves are used as the training set and predictions are made on the remaining. We borrow the results from (Bande and Fuente, 2012) and they, along with the results from the FNN, are given in Table

3.2. In the original paper, the authors use the metric *mean squared error of prediction*, MEP $= \frac{\text{MSPE}}{\text{Var}(y)}$, where MSPE is the average squared error of the test set and $\text{Var}(y)$ is the variance of the true response (we can think of MEP as a rescaling of the MSPE) to assess the models. They also used R^2, which we tabulate in Table 3.2. In the functional neural network, we tuned to find that a six-layer network was optimal with a total of 4029 parameters. The exact configuration is provided in the Appendix. We found that our model has the lowest MEP, but is about 3% lower than the best R^2. Most other models perform worse with the *Semi-Functional Partial Linear Model* (Aneiros-Pérez and Vieu, 2006) being the most comparable.

3.3 Canadian Weather Data

The data set used here has information regarding the total amount of precipitation in a year and the daily temperature for 35 Canadian cities. We are interested in modelling the relationship between precipitation and temperature. Generally, you would expect that lower temperatures would indicate higher precipitation rates. However, this is not always the case. In some regions, the temperature might be very low, but the inverse relationship with rain/snow does not hold. Our goal is to see whether we can successfully model these anomalies relative to other methods.

Model	MSPE_{CV}	R^2
Functional Linear Model (Basis)	0.123	0.00312
Functional Non-Parametric Regression	0.0561	0.0900
Functional PC Regression	0.0272	0.352
Functional PC Regression (2nd Deriv Penalization)	0.0930	0.00298
Functional PC Regression (Ridge Regression)	0.0259	0.382
Functional Partial Least Squares	0.0449	0.177
Functional Partial Least Squares (2nd Deriv Penalization)	0.0483	0.155
Neural Networks	0.126	0.0453
Functional Neural Networks	**0.0194**	**0.541**

Table 3.3: Tabulated leave-one-out cross-validated mean-squared predication error and R^2 of eight models, including the FNN. The italics indicate second-best model performance, whereas the bolded (and green coloured) cells indicate best performance. We see that the *functional neural network* performed the best.

The functional observations are defined for the temperature of the cities for which there are 365 (daily) time points, t. In total, there are 35 functional observations and the scalar response is the average precipitation. A Fourier basis expansion was used with 65 basis functions defining each of the 35 cities (Ramsay et al., 2009). The choice of the number of basis functions defining the FNC is left to tuning. The results from two criteria ($R^2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2}$ and a leave-one-out-cross-validated MSPE) are measured for a number of models. We see that the FNN model outperforms all other approaches including the usual neural networks. All models were tuned with the final choice for the FNN being presented in the Appendix. Table 3.3 summarizes the predictive results.

We can observe the recovered coefficient functions in Figure 3.2. We compare the recovered coefficient from the functional linear model with the FNC. We observe similar patterns between the two. Note that the difference between the two only accounts for some of the difference in R^2. The FNN has many more parameters allowing for more flexibility in the modelling process and thus the great increase in accuracy. With respect to the number of basis terms for the coefficient function, we optimized to find the optimal number for the linear model (11) and used the same number for the network. This was to measure how similar the coefficient curves would be under the same conditions.

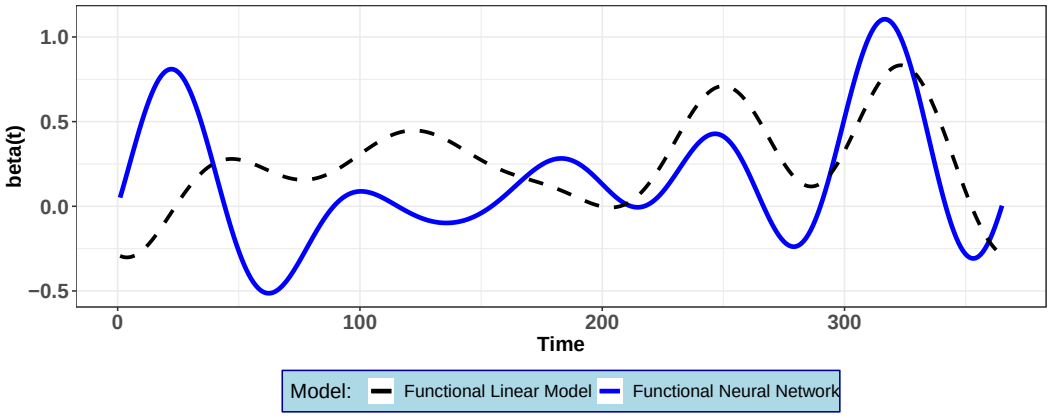

Figure 3.2: The estimated coefficient functions for the usual functional linear model and the functional neural network for the weather data set. For this data set, we decided to keep the number of basis functions the same across both models - the choice for this was 11 and comes from (Ramsay et al., 2009).

Chapter 4

Simulation Studies

4.1 Recovery of $\beta(s)$

In this section, we present results from when we know the true underlying coefficient function. The goal is to compare the functional neural coefficient to the coefficient function extracted using functional regression. Useful results here would go a long way in showing that the FNN is not only useful for prediction, but can be a valiant tool when the goal is to approximate relationships via parameter estimation. In order to measure this, the integrated mean square error (IMSE) is used which is defined as

$$\text{IMSE} = \frac{1}{|\mathcal{S}|}\int_{\mathcal{S}}(\beta(s) - \hat{\beta}(s))^2 ds,$$

where $\hat{\beta}(s)$ is the predicted coefficient function either from the FNN or from the functional regression. We use the following to generate our response, y:

$$y^* = g\left(\alpha + \int_{\mathcal{S}} \beta(s)(\sum_i \phi_i \psi_i(s))\,\mathrm{d}s\right) + \epsilon^*, \tag{4.1}$$

where our choice for β is

$$\beta(s) = m_1 + m_2\sin(s\pi) + m_3\cos(s\pi) + m_4\sin(2s\pi) + m_5\cos(2s\pi),$$

α_i is sampled from the uniform distribution, $X \sim \mathcal{U}(d, e)$ for the i^{th} observation, and ϵ^* is sampled from the Gaussian distribution, $Y \sim \mathcal{N}(0, 1)$. The true data from which $x(s)$ is generated comes from either $a \cdot sin(a) + b$ or $c \cdot \exp(a) + sin(a) + b$, where $a \sim \mathcal{N}(0, 1)$,

$b \sim \mathcal{N}(0, \frac{i}{100})$, and $c \sim \mathcal{N}(0, 1)$ are parameters that govern the difference between the functional observations.

This generative procedure will be used for four different simulations. In all four, we generate 300 observations randomly using (4.1) by varying a, b and c. The coefficients for $\beta(s)$ are set beforehand. We fit the functional linear model and the functional neural network for 250 iterations of these 300 generations of data. We cross-validate over a grid for λ in order to find a smooth (and less volatile) estimate of $\beta(s)$ from the functional linear model. The difference is measured using IMSE.

The first simulation is for when the link function $g(\cdot)$ is the identity function. Here, we would expect the functional linear model to perform comparably (if not better) than the functional neural network due to its deterministic nature and the linear relationship. In the second simulation, we look to see if our method can recover $\beta(s)$ for when the link function is exponential. The third simulation explores this behaviour for a sigmoidal relationship. And lastly, we simulate a logarithmic relationship between the response and the functional covariates. These simulations are summarized as follows:

$$
\begin{aligned}
&\text{Simulation 1} : y^* = \alpha + \int_{\mathcal{S}} \beta(s)(\sum_i \phi_i \psi_i(s))\, \mathrm{d}s + \epsilon^* \\
&\text{Simulation 2} : y^* = \exp\left(\alpha + \int_{\mathcal{S}} \beta(s)(\sum_i \phi_i \psi_i(s))\, \mathrm{d}s\right) + \epsilon^* \\
&\text{Simulation 3} : y^* = \frac{1}{1 + \exp\left(\alpha + \int_{\mathcal{S}} \beta(s)(\sum_i \phi_i \psi_i(s))\, \mathrm{d}s\right)} + \epsilon^* \\
&\text{Simulation 4} : y^* = \log\left(\left|\alpha + \int_{\mathcal{S}} \beta(s)(\sum_i \phi_i \psi_i(s))\, \mathrm{d}s\right|\right) + \epsilon^*.
\end{aligned}
$$

In all but the fourth scenario, we use a three-layer network with *ReLU* and *linear* activation functions. In the final scenario, we use a one-layer hidden network with a sigmoidal activation function. With respect to the linear model, we cross-validate over a grid to find the optimal λ parameter to smooth the resulting coefficient function. In Figure 4.1, we present the results for these four simulations. We observe that the usual linear model seems to perform better when the relationship is linear. There are far more parameters in the FNN that are contributing to the prediction of y. When the relationship is non-linear, the functional linear

model struggles where relatively, the FNN does a much better job in recovering the true $\beta(s)$. Note that, in the case of more than a single neuron in the functional layer, we take an average of the basis coefficient estimates across the n_1 neurons. The averages of these results along with computation times, are provided in Table 4.1.

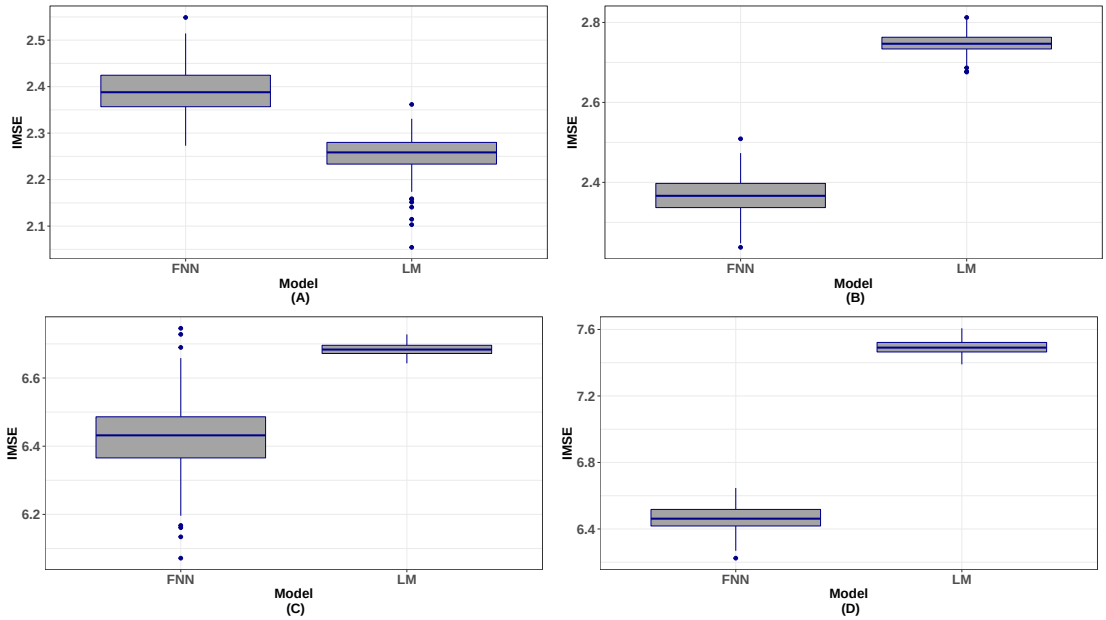

Figure 4.1: Boxplots of root IMSE results over 250 simulation runs for four scenarios. Plot (A) is for when we use the identity link function. Plot (B) are the results from the exponential link. The bottom plots, (C) and (D) are the results from simulation 3 and 4, respectively.

	Functional Linear Model			Functional Neural Networks		
	Mean	SD	Avg. Comp. Time	Mean	SD	Avg. Comp. Time
Simulation: 1	5.081	0.1471	0.2319s	5.705	0.2403	4.679s
Simulation: 2	7.549	0.1292	0.2322s	5.609	0.2195	4.672s
Simulation: 3	44.68	0.2367	0.2470s	41.33	1.335	5.999s
Simulation: 4	56.17	0.6502	0.2580s	41.77	0.9566	6.302s

Table 4.1: Information regarding simulation runs. The average values along with the associated deviation of the IMSE over the 250 iterations are provided for both the FNNs and the *functional linear model*. The computation times given are the average per simulation.

4.2 Prediction

We can also observe how our method predicts the response y^*, relative to other functional and multivariate approaches across the simulations, and under the four different conditions given in the previous section. We are interested in seeing how FNNs perform versus

functional and multivariate approaches. The multivariate methods to be compared include: *least squares regression* (MLR), *LASSO* (Tibshirani, 1996), *random forests* (RF) (Breiman, 2001), *gradient boosting* approaches (GBM, XGB) (Friedman, 2001) (Chen et al., 2015), and *projection pursuit regression* (PPR) (Friedman and Stuetzle, 1981).

We did not tune our network (and in fact, we kept the same configuration for the FNNs that we had in the previous simulations for the four variations) but made an effort to tune all the other models. For example, the choice of λ for the *LASSO* was made using cross-validation. The tree methods were tuned for across a number of their hyperparameters (such as the amount of parameters [*mtry* (Liaw and Wiener, 2002)] chosen for each built tree and the number of nodes), and for PPR, we built models with a number of terms and picked the model with the lowest MSPE. In these simulations, we did 100 runs. For each run, we generated 300 functional observations with a corresponding value of y^* in accordance to (4.1). After the realization, we split the data randomly, built a model on the training set, and predicted on the test set. This process is repeated for the same four simulation scenarios as given in 4.1. The box plots in Figure 4.2 measure the relative error in each simulation run. We call this the relative MSPE,

$$\text{rMSPE} = \frac{\text{MSPE}}{\min\limits_{\text{all models}} \text{MSPE}}.$$

For example, on any given run, we calculate the MSPE values for each model, and then divide each of them by the minimum in that run. The best model according to this measure will have a value of 1. Averages centered at values greater than 1 perform worse. The results of the simulations are summarized in Figure 4.2 (also see Table D.1 in the supplementary materials for the absolute MSPE).

The relative measure we use makes it easy to compare each model within a simulation, and across the four simulation classes. Notably, we see in Figure 4.2 that the FNN performs well, not only within a simulation, but also we see it perform comparably to itself among the different simulations. This can be attributed to the addition of functional information passed into the network, so that the learning efficiency increases. By learning efficiency we mean that for a given training data sample (here, a curve), the network can learn more

about the underlying distribution in one epoch. Therefore, we can learn more about the underlying distribution function, without crossing over to an over-fitting regime. In the curve building process, we assume that there is noise associated with the observed discrete values - by reverse engineering into an approximation of the curve, we effectively reduce that noise and then later, avoid some of the error chasing that we would otherwise be privy to. This is a good application of Theorem 1, as we proved that this method should produce estimates of the response that come arbitrarily close to the true response (given that the response is a continuous function). As a comparison, we see that generally the tree-based methods perform comparably within a particular simulation, but performance changes across different simulations. We expect this behaviour from these methods because there are underlying assumptions about the space of functions that tree-based methods can learn (without over-fitting completely). With respect to the outliers, we can see that they are most prevalent in *Simulation 2*; this is because this simulation was for when the link function $g(\cdot)$ was exponential. In this context, we expect that the raw difference between our prediction and the true value is greater than it would be in the other cases - this leads to a plethora of outliers observed.

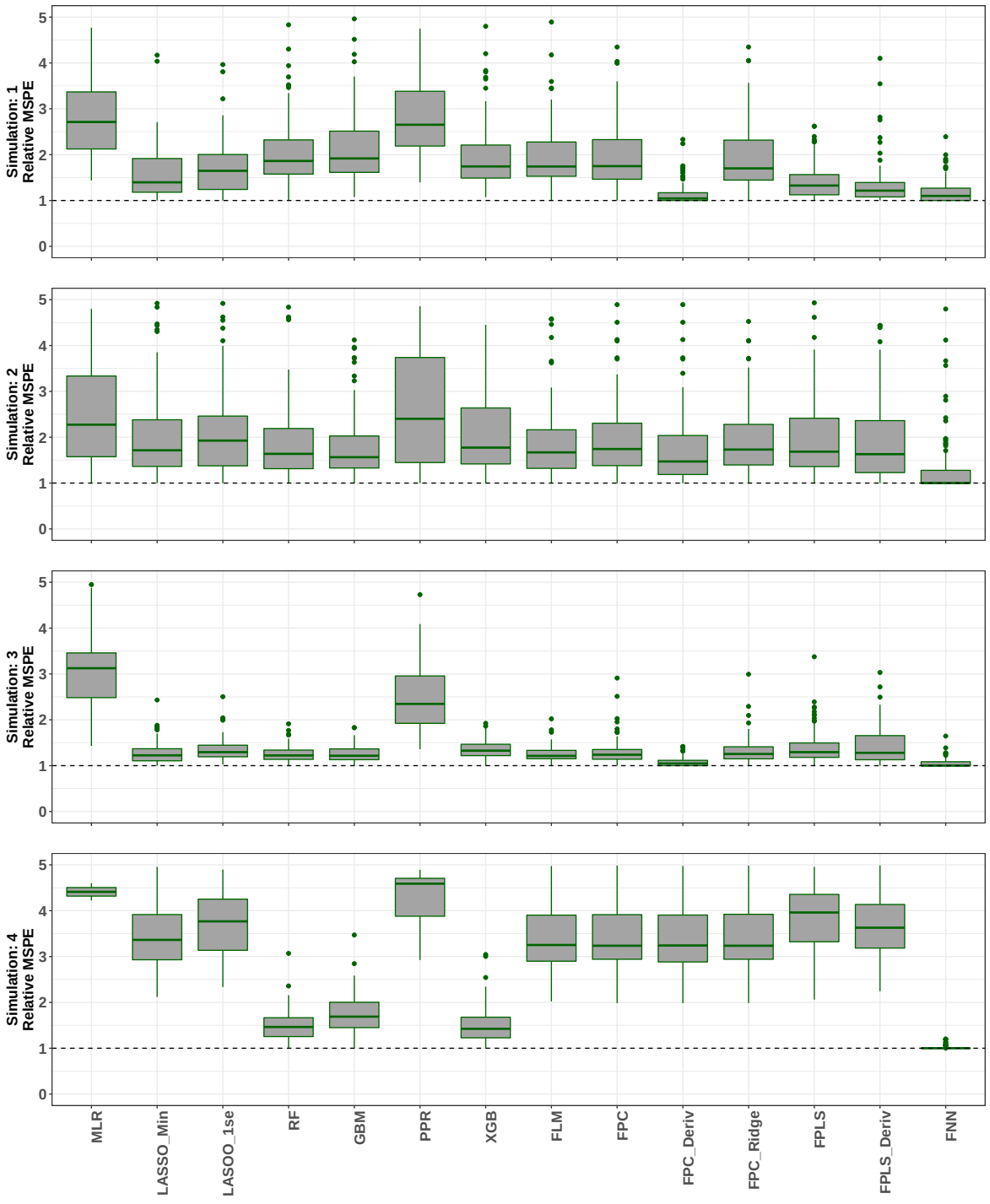

Figure 4.2: Boxplots of rMSPE values for all simulations.

Chapter 5

Conclusions & Discussions

The extreme rise in popularity of deep learning research has resulted in enormous breakthroughs in computer vision, classification, and scalar prediction. However, these advantages thus far had been limited to when the data is treated as discrete. This paper introduced the first of a family of neural networks that extend into the functional space.

In particular, we present a functional feed-forward network for when the responses are scalar. We developed a methodology which showed the steps required to compute a solution for the network. This methodology took advantage of integration approximation methods and the usual gradient descent approach. Multiple examples were provided which showed that the functional neural network outperformed a number of other functional models and multivariate methods with respect to the mean squared prediction error. It was also shown through simulation studies that the recovery of the true underlying coefficient function is better done by the FNN than the functional linear model when the relationship is non-linear.

To extend this project, algorithms can be developed for other combinations of input and output types such as the function on function case (Wang et al., 2016). Moreover, one can consider adding additional constraints to the first-layer neurons via penalization or other methods.

Bibliography

Manuel Bande and Manuel Fuente. Statistical computing in functional data analysis: The r package fda. usc. 2012.

James O. Ramsay, Giles Hooker, and Spencer Graves. *Functional data analysis with R and MATLAB*. Springer New York, 2009.

Jim Ramsay and B. W. Silverman. *Functional data analysis*. Springer New York, 2010.

Hervé Cardot, Frédéric Ferraty, and Pascal Sarda. Functional linear model. *Statistics & Probability Letters*, 45(1):11–22, 1999.

Hervé Cardot, Frédéric Ferraty, and Pascal Sarda. Spline estimators for the functional linear model. *Statistica Sinica*, 13(3):571–591, 2003.

Trevor Hastie and Colin Mallows. A statistical view of some chemometrics regression tools: Discussion. *Technometrics*, 35(2):140–143, 1993.

Hans-Georg Müller, Ulrich Stadtmüller, et al. Generalized functional linear models. *the Annals of Statistics*, 33(2):774–805, 2005.

Ci-Ren Jiang, Jane-Ling Wang, et al. Functional single index models for longitudinal data. *The Annals of Statistics*, 39(1):362–388, 2011.

Cristian Preda, Gilbert Saporta, and Caroline Lévéder. Pls classification of functional data. *Computational Statistics*, 22(2):223–235, 2007.

Frédéric Ferraty and Philippe Vieu. *Nonparametric functional data analysis: theory and practice*. Springer Science & Business Media, 2006.

Germán Aneiros-Pérez and Philippe Vieu. Semi-functional partial linear regression. *Statistics & Probability Letters*, 76(11):1102–1110, 2006.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 2012.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

George AF Seber and Alan J Lee. *Linear regression analysis*, volume 329. John Wiley & Sons, 2012.

Robert Tibshirani, Trevor Hastie, and Jerome Friedman. *The elements of statistical learning: data Mining, inference, and prediction*. Springer, 2009.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 2011.

Jun Han and Claudio Moraga. The influence of the sigmoid function parameters on the speed of backpropagation learning. In *International Workshop on Artificial Neural Networks*. Springer, 1995.

Guido Fubini. Sugli integrali multipli. *Rend. Acc. Naz. Lincei*, 16:608–614, 1907.

George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

David Rumelhart, Geoffrey Hinton, and Ronald Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

YK Kim and JB Ra. Weight value initialization for improving training speed in the backpropagation network. In *IEEE International Joint Conference on Neural Networks*. IEEE, 1991.

Mercedes Fernández-Redondo and Carlos Hernández-Espinosa. Weight initialization methods for multilayer feedforward. In *ESANN*, 2001.

François Chollet et al. Keras. `https://keras.io`, 2015.

Hadi Fanaee-T and Joao Gama. Event labeling combining ensemble detectors and background knowledge. *Progress in Artificial Intelligence*, 2(2-3):113–127, 2014.

Hans Henrik Thodberg. Tecator meat sample dataset. statlib datasets archive, 2015.

Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.

Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

Jerome Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, 29(5):1189–1232, 2001.

Tianqi Chen, Tong He, Michael Benesty, Vadim Khotilovich, and Yuan Tang. Xgboost: extreme gradient boosting. *R package version 0.4-2*, pages 1–4, 2015.

Jerome Friedman and Werner Stuetzle. Projection pursuit regression. *Journal of the American statistical Association*, 76(376):817–823, 1981.

Andy Liaw and Matthew Wiener. Classification and regression by randomforest. *R News*, 2(3):18–22, 2002. URL `https://CRAN.R-project.org/doc/Rnews/`.

Jane-Ling Wang, Jeng-Min Chiou, and Hans-Georg Müller. Functional data analysis. *Annual Review of Statistics and Its Application*, 3:257–295, 2016.

Appendix A

Proof of Theorem 1.

Theorem 1. *Let $g : \mathbb{R} \to \mathbb{R}$ be any continuous sigmoidal function, I_n denote the $n-$dimensional hypercube $[0,1]^n$ and $\mathcal{C}(I_n)$ denote the space of continuous functions. Then, the finite sum of the following form is dense in $\mathcal{C}(I_n)$:*

$$h(s) = \sum_{n=1}^{n_1} \Psi_n g\left(\sum_{k=1}^{K}\left(\int_{\mathcal{S}} \beta_{nk}(s) x_k(s)\,\mathrm{d}s\right) + \sum_{j=1}^{J} w_{nj} z_j + b_n\right),$$

meaning that for any $f(s) \in \mathcal{C}(I_n)$ and for $\epsilon > 0$, the function $h(s)$ obeys:

$$|h(s) - f(s)| < \epsilon$$

Proof. The proof is to show that $h(s)$ has the form from Cybenko's result, *i.e.* Cybenko form (Cybenko, 1989):

$$G(x) = \sum_{n=1}^{N} \alpha_n \sigma\left(y_n^T z + \theta_n\right) = \sum_{n=1}^{N} \alpha_n \sigma\left(\sum_{j=1}^{J} y_{nj} z_j + \theta_n\right).$$

To begin we consider our function g and its argument,

$$g\left(\sum_{k=1}^{K}\left(\int_{\mathcal{S}} \beta_{nk}(s) x_k(s)\,\mathrm{d}s\right) + \sum_{j=1}^{J} w_{nj} z_j + b_n\right),$$

and we note that

$$\sum_{j=1}^{J} w_{nj} z_j + b_n,$$

is already in Cybenko form. Therefore, all we need to show is that

$$\sum_{k=1}^{K}\left(\int_{\mathcal{S}} \beta_{nk}(s) x_k(s)\,\mathrm{d}s\right),$$

depends only on n, the neuron number index. If we expand our weight function as a finite linear combination of basis functions, we get:

$$\beta_{nk}(s) = \sum_{m=1}^{M_k} c_{nkm}\phi_{nkm}(s),$$

Therefore,

$$\begin{aligned}
\sum_{k=1}^{K}\left(\int_{\mathcal{S}} \beta_{nk}(s)x_k(s)\,\mathrm{d}s\right) &= \sum_{k=1}^{K}\left(\int_{\mathcal{S}} \sum_{m=1}^{M_k} c_{nkm}\phi_{nkm}(s)x_k(s)\,\mathrm{d}s\right) \\
&= \sum_{k=1}^{K}\sum_{m=1}^{M_k} c_{nkm}\underbrace{\left(\int_{\mathcal{S}} \phi_{nkm}(s)x_k(s)\,\mathrm{d}s\right)}_{a_{nkm}} \\
&= \sum_{k=1}^{K}\sum_{m=1}^{M_k} c_{nkm}a_{nkm} \\
&= \tilde{b}_n
\end{aligned}$$

Note, since we have a finite sum under the integral, we can integrate term-by-term and switch the integral and sum in the second step above. In total, we have:

$$g\left(\sum_{k=1}^{K}\left(\int_{\mathcal{S}} \beta_{nk}(s)x_k(s)\,\mathrm{d}s\right) + \sum_{j=1}^{J} w_{nj}z_j + b_n\right) = g\left(\sum_{j=1}^{J} w_{nj}z_j + \left(b_n + \tilde{b}_n\right)\right)$$

Therefore, if we let $\alpha_n = \Psi_n$, $y_{nj} = w_{nj}$, and $\theta_n = b_n + \tilde{b}_n$, we obtain Cybenko's form for $g(\cdot)$. ■

Appendix B

List of All Parameters

Parameter	Type	Details
$\beta(s)$	Estimated	Coefficient function found by the FNN.
w	Estimated	The scalar covariate weights.
b	Estimated	The bias in each neuron.
Number of Layers	Hyperparameter	The *depth* of the FNN.
Neurons per Layer	Hyperparameter	Number of neurons in each layer of the FNN.
γ	Hyperparameter	The learning rate of the FNN.
Decay Rate	Hyperparameter	A weight on the learning process across epochs for the FNN.
Validation Split	Hyperparameter	The split of train/test set.
FNC Basis	Hyperparameter	The size of M for the estimation of $\beta(s)$.
Epochs	Hyperparameter	The number of learning iterations.
Batch Size	Hyperparameter	Subset of data per pass of the FNN.
Activations	Hyperparameter	The choice of $g(\cdot)$ for each layer.
Early Stop	Hyperparameter	Stops the model building process if no improvement in error.

Table B.1: A list of the parameters in the network.

Appendix C

Model Hyperparameter Values

Model	Layers	Neurons	Activations	FNC Basis	Learn Rate
Weather	2	c(16, 8)	c(relu, sigmoid)	5	0.05
Bike	4	c(32, 32, 32, 32)	c(sigmoid, sigmoid, relu, linear)	3	0.002
Tecator	6	c(24, 24, 24, 24, 24, 58)	c(relu*5, linear)	3	0.005
Sim 1 Rec	3	c(16, 16, 16)	c(relu, linear, linear)	5	0.001
Sim 2 Rec	3	c(16, 16, 16)	c(relu, linear, linear)	5	0.001
Sim 3 Rec	1	c(16)	c(sigmoid)	5	0.01
Sim 4 Rec	3	c(16, 16, 16)	c(relu, linear, linear)	5	0.001
Sim 1 Pred	3	c(16, 16, 16)	c(relu, linear, linear)	5	0.001
Sim 2 Pred	3	c(16, 16, 16)	c(relu, linear, linear)	5	0.001
Sim 3 Pred	1	c(16)	c(sigmoid)	5	0.01
Sim 4 Pred	3	c(16, 16, 16)	c(relu, linear, linear)	5	0.001

Table C.1: Configurations for FNN models throughout the paper.

Appendix D

MSPE Values for Simulated Data

Sim/Model	Simulation 1	Simulation 2	Simulation 3	Simulation 4
Functional Linear Model (Basis)	0.2325	0.9448	0.01788	0.3933
Functional PC Regression	0.2320	0.9694	0.01873	0.3933
Functional PC Regression (2nd Deriv Penalization)	0.1228	0.8610	0.01560	0.3923
Functional PC Regression (Ridge Regression)	0.2318	0.9680	0.01884	0.3933
Functional Partial Least Squares	0.1599	0.9536	0.02035	0.6552
Functional Partial Least Squares (2nd Deriv Penalization)	0.1440	0.9126	0.02058	0.6107
Functional Neural Networks	0.1397	0.7063	0.01518	0.1218
Multiple Linear Regression	0.4582	1.373	0.06553	2.047
LASSO - Min λ	0.1704	1.009	0.01830	0.4017
LASSO - 1SE λ	0.1961	1.043	0.01930	0.4450
Random Forest	0.2660	0.9618	0.01795	0.1704
Gradient Boosting	0.2799	0.9727	0.01792	0.1967
Projection Pursuit Regression	0.3625	1.370	0.03617	1.229
Extreme Gradient Boosting	0.2556	1.098	0.01934	0.1687

Table D.1: MSPE values for simulated data predictions. As it was evident by the boxplots in Figure 6, the FNN approach outperforms the others in 3 of the 4 simulations. In the one case that it does not, it performs second best.