Functional Neural Networks
Prediction & Inference

Barinder Thind[*]
Department of Statistics & Actuarial Science
Simon Fraser University
Burnaby, British Columbia
bthind@sfu.ca

Abstract

This paper introduces a methodology for applying feed-forward neural networks when the initial activations are of a functional nature. The model is specific for scalar responses with at least one functional covariate. An extension is then presented which serves as the functional analogue to residual neural networks; this extension leads to a key insight which remedies the traditionally non-inferential nature of neural networks. The model is shown to outperform the usual neural networks and functional linear regression.

Keywords: Functional Data Analysis, Gaussian Quadrature, Black-Box Models, Neural Networks, Residual Neural Networks

1 Introduction

The ever-expanding umbrella that encompasses deep-learning methodologies has been limited to the multivariate scope. With the advent and rise of *functional data analysis* (FDA) [11], it is natural to extend neural networks to this (functional) space. More specifically, this paper introduces functional feed-forward neural networks and functional residual neural networks for scalar responses with functional covariates; the latter of which allows for a type of inference to be made that has not been possible thus far in the multivariate case.

Neural networks have primarily been used for and have excelled at prediction/classification - this success has come at the cost of inference. Classical linear regression models have a relatively clear interpretation of the parameters estimates[2] - however, this interpretation does not carry through to the culmination of generalized linear models that the neural network is[3]. In the functional linear regression case, the coefficient parameters being estimated are functions, $\beta(t)$ rather than a set of scalar values, $\vec{\beta}$. This paper details an approach that makes the functional coefficient traditionally found in the regression model, readily available from the neural network procedure and thus, allowing for inference from a set of models with superior predictive power.

In Section 2 and Section 3, a brief introduction to neural networks and FDA is provided, respectively. In Section 4, *functional neural networks* (FNNs) are introduced for the feed-forward and residual network case. Additionally, commentary is provided on the inferential potential of these new networks.

[*] https://b-thi.github.io
[2] Given the linear model assumptions hold
[3] Assuming that the number of layers is > 1

Then, Section 5 provides results for the aforementioned models. Lastly, Section 6 contains some closing thoughts and extensions for the future for this new set of models.

2 Neural Networks

In this section, I highlight the underlying methodology of neural networks. I begin by presenting the "forward pass" and then provide details on how the model "learns" through gradient descent.

2.1 Forward Pass

For simplicity sake, we will first look at a network with just a "singe hidden layer". Consider Figure 1 - the blue *"neurons"* contain values of the input data. For example, if the input was an image, then each neuron would hold a value corresponding to the gray scale value of each pixel of the image. This is known as the *activation* value. The red neurons contained within the second row are referred to as neurons from the *hidden layer*. Each single layer is a non-linear transformation of a linear combination of each activation in the first layer. For example, $z^{(1)}$ would be defined as:

$$z^{(1)} = \sigma(\vec{\alpha_{0_1}} + \vec{x}\boldsymbol{\alpha^{(1)}}) \tag{1}$$

Where $\boldsymbol{\alpha^{(1)}}$ and $\vec{\alpha_{0_1}}$ is the set of weights and biases[4] and $\sigma()$ is some *activation function* that transforms the resulting linear combination (for example, we might want probabilities for a binary response so we could use the sigmoid function[5] when that is the case). Note that the vector $\vec{x}$ corresponds to a single "row" of our data set (or, a single image if that was the data type we were working with) and is therefore a p-dimensional vector where p is the number of covariates.

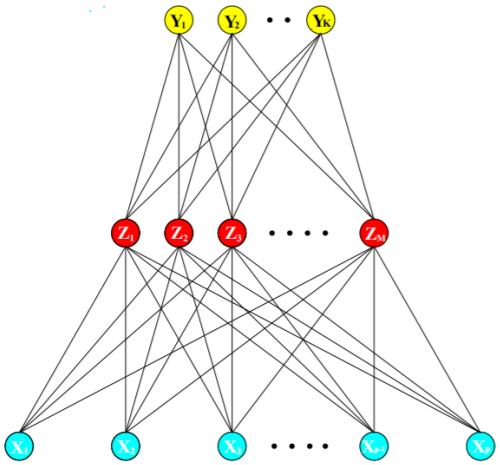

Figure 1: An overview of a single hidden layer network [12]

Once we have the values for the m[6] neurons in the hidden layer, we have another set of activations. Using these, we can move onto the final layer. In the case of image classification, say for the purposes of handwriting recognition, an image might correspond to a single number $y \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. In this case, $K = 10$ and is the number of possibilities for classification. The last transformation assigns a probability to each of the K classes effectively providing a likelihood that your observation

[4] These are initialized randomly in this simple case

[5] $\frac{1}{1+e^{-x}}$

[6] The choice of M is left to the user

(in our example, the single image) belongs to each of them, respectively. Often, the softmax function:

$$g_k(T) = \frac{e^{T_k}}{\sum_{i=1}^{K} e^{T_i}} \tag{2}$$

is used as it allows probability specification for a number of classes[7]. Here, there are k sets of T values and these are all linear combinations moving from the hidden layer to the output layer. You can imagine that this could be generalized to n number of hidden layers with some choice of m neurons in each of them resulting in a myriad of parameters[8] to be estimated.

2.2 Backpropogation

The "learning" of this approach takes place in a process called *backpropogation*. In order to move forward, we must first define a loss function. This function is some measure of the aggregated residual between our prediction and the true value. One approach is to use squared-error:

$$R(\theta) = \sum_{k=1}^{K} \sum_{i=1}^{N} (y_{ik} - f_k(x_i))^2 \tag{3}$$

This function is a sum of the difference between the prediction for every observation for every output. For example, the outer sum would be of the difference between the predicted probability of an image belonging to a particular class, over all classes and the inner sum would aggregate over every image (or observation) you have[9].

Now that we have a measure of error, we can look to minimize it. This function takes in $(p+1) \cdot M \cdot (M+1) \cdot K$ parameters[10] resulting in a high dimensional gradient. This translates to an inordinate amount of peaks and valleys on the optimization landscape. It is also very likely that the global minimizer of $R(\theta)$ will overfit the data so any local minimizer may serve us better; in fact, we will take small steps towards the optimum specified by a "learning rate", γ.

For simplicity sake, we consider a network with a single neuron in its 2 hidden layers and only look at a 1-dimensional observation [2].

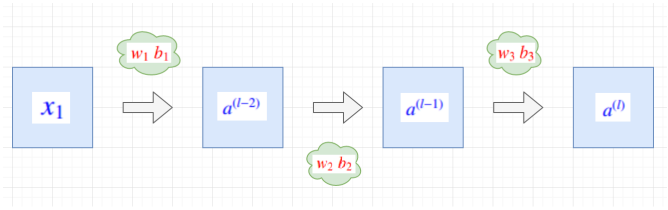

Figure 2: Schematic representing toy example

Then, the cost function, R will have 6 parameters outlined in red in Figure 2. Using (3), we see that in this simple case, the cost function reduces to $R_1 = (z^{(l)} - y)^2$ where $a^{(l)}$ is the activation in the final layer[11] defined as: $a^{(l)} = \sigma(w^{(l)}a^{(l-1)} + b^{(l)})$. For convenience, define $w^{(l)}a^{(l-1)} + b^{(l)}$ as $u^{(l)}$. Remember, we want to minimize the cost function; we can note that a change in the weight $w^{(l)}$ causes some change to the cost function, R_1 - we want to know this change: $\frac{\partial R_1}{\partial w^{(l)}}$ as our goal is to minimize this. Using chain rule, we can observe that this derivative can be broken down to a number

[7] In fact, letting $k \in \{0, 1\}$ returns the famous logistic transformation
[8] These being the weights and biases of the model
[9] There are a plethora of loss functions to pick from; for example, cross-entropy and log-loss
[10] The set θ encompasses these parameters
[11] Let l be indicative of the last layer i.e. $w_3 = w^{(1)}$ and $w_2 = w^{(l-1)}$

of sub-derivatives:

$$\frac{\partial R_1}{\partial w^{(l)}} = \frac{\partial u^{(l)}}{\partial w^{(l)}} \cdot \frac{\partial a^{(l)}}{\partial z^{(l)}} \cdot \frac{\partial R_1}{\partial a^{(l)}} \tag{4}$$

$$= a^{(l-1)} \cdot \sigma'(z^{(l)}) \cdot 2 \cdot (a^{(l)} - y) \tag{5}$$

We want to find the roots of this derivative but, not only of *this* derivative. First, we note that (3) is defined for every observation:

$$\frac{\partial R}{\partial w^{(l)}} = \frac{1}{n} \sum_{i=0}^{n-1} \frac{\partial R_i}{\partial w^{(l)}} \tag{6}$$

And note that (6) is only one of the 6 derivatives making up the gradient of the cost function:

$$\nabla R = \left(\begin{array}{cccccc} \frac{\partial R}{\partial w^{(l)}} & \frac{\partial R}{\partial w^{(l-1)}} & \frac{\partial R}{\partial w^{(l-2)}} & \frac{\partial R}{\partial b^{(l)}} & \frac{\partial R}{\partial b^{(l-1)}} & \frac{\partial R}{\partial b^{(l-2)}} \end{array} \right)^T = \vec{0}$$

The average of the parameter values that satisfy the above equation are the changes we need to make to the current weights. The change is done proportional to the aforementioned learning rate, γ. For example, the update for some parameter w_3 would be $w_3 = w_3 + \frac{\partial R}{\partial_{w_3}} \cdot \gamma$. Optimal values are found in *mini batches*; these are subsets of observations for which the optimization takes place as opposed to the entire data set[12]. This route is taken due to computational feasibility. The process repeats for some number of *epochs*[13] after which, the model has been "trained".

3 Functional Data

Often, we have repeated measurements of some number of individuals over some continuum, **S**. In the usual perspective, this data would be seen as discrete values. However, this data can be treated as a single *functional* observation rather than as a collection of multivariate data. This approaoch allows for a number of advantages; for example, the ability to take derivatives to infer how the underlying function changes over the continuum. An illustration is provided in Figure 3.

Basis functions are used to represent functions. Think of observed discrete data points as single entities. Functional is a reference to the intrinsic nature of the data. Functional data is also observed and recorded as a pair: (s_j, y_j) - this can be seen as a snapshot of the function at some value of the continuum s. Let $x(s)$ be the functional observation to be estimated. We assume that $x(s)$ has reason to exist in the given context. A smooth $x(s)$ will indicates that derivatives exist allowing us to estimate various dynamics of the function for example, we could estimate position, velocity, and acceleration if we had observations for such data.

In general, we are concerned with a collection or sample of functional data, rather than one single functional observation, $x(s)$.

$$x_i = (s_{ij}, y_{ij}) = \begin{bmatrix} (s_{11}, s_{11}) & (s_{12}, s_{12}) & \dots & (s_{1n}, s_{1n}) \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ (s_{m1}, s_{m1}) & \dots & \dots & (s_{mn}, s_{mn}) \end{bmatrix}$$

Each row above can be thought of as a separate functional observation to be estimated. Note that sometimes the s values might be the same across all discrete data points, hence the notation can be simplified.

[12] This approach is also known as *stochastic gradient descent*

[13] A single epoch is one full forward and backward pass for every observation in your data set

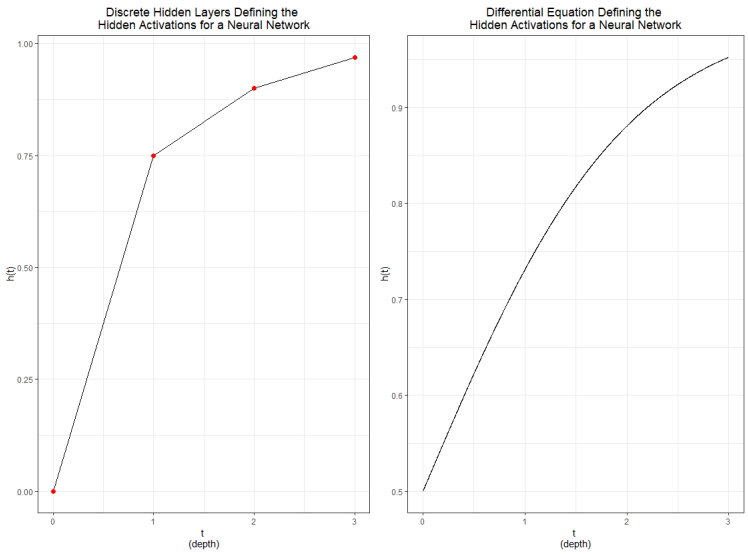

Figure 3: This image of depth in a neural network illustrates a comparison of the lens through which we can view data. The red dots can be viewed as observed data and the curve on the right is the underlying functional process that generates the data - one of the goals of FDA is to estimate this process.

Since our estimation will likely not be exact interpolation of the discrete data points, the assumption is made that the difference between the functional fit and the discrete value is a matter of random error (corresponding to the "roughness" of the data). That is:

$$y = x(s) + \epsilon \tag{7}$$

Where:

$$x(s) = \sum_{k=1}^{K} c_k \phi_k(s) \tag{8}$$

$$= c_1\phi_1(s) + c_2\phi_2(s) + ... + c_K\phi_K(s) \tag{9}$$

$$= c^T\phi \tag{10}$$

So how then, do we calculate the coefficient c^T? One method to do so is to use the classic ordinary least squares estimation. If we defined Φ to be the n by K matrix containing our K basis functions each evaluated at the values of s, then the estimate for c_k is:

$$\hat{c} = (\Phi^T\Phi)^{-1}\Phi^T y \tag{11}$$

And therefore, $\hat{y} = \Phi(\Phi^T\Phi)^{-1}\Phi^T y$. This is called the least squares smoother in the context of functional data analysis. However, the classic model may not always be realistic for the data we have due to autocorrelation and non-stationarity of the residuals and therefore, alternative approaches are much better suited [11].

The focus here will be on the first. The usual functional linear model for a scalar response and functional covariate is notated as follows:

$$E(Y|X) = \alpha_0 + \int_0^S \beta(s)x_i(s)ds \tag{12}$$

Where α_0 is the intercept term and you could think of $\beta(t)$ as a function which generates the coefficients associated with the co-variate at any point from 0 to S. Also, $x_i(s)$ is the i^{th} functional functional observation used to predict the scalar response y.

In order to estimate $\beta(s)$, the error is minimized:

$$\beta(s) = argmin\left\{\sum(y_i - \alpha_0 - \int_0^T \beta(s)x_i(s)ds)^2\right\} \tag{13}$$

However, this can result in volatile estimates along with some significant probability of over-fitting. Instead then, a constrained approach is used:

$$PENSSE_\lambda(\beta) = argmin\left\{\sum(y_i - \alpha_0 - \int_0^S \beta(s)x_i(s)ds)^2 + \lambda\int[\beta(s)]^2ds\right\} \tag{14}$$

Note: we still represent $\beta(s)$ in some functional form $\beta(s) = \sum c_i\phi_i(s)$. Now, let $Z_i = [1, \mathbf{x}_i]$ and $\mathbf{x}_i = \int \Phi(s)x_i(s)ds$. Then, we can rewrite the model as:

$$\mathbf{y} = \mathbf{Z}\begin{bmatrix}\alpha & \mathbf{c}\end{bmatrix}^T + \epsilon \tag{15}$$

Essentially then, the vector in the above equation is what we need to estimate. Under the smoothing conditions, the estimate is:

$$[\hat{\alpha}\ \hat{c}^T]^T = (Z^TZ + \lambda R_L)^{-1}Z^T\mathbf{y} \tag{16}$$

4 Functional Neural Networks

4.1 A Feed-Forward Approach: Scalar Response, Functional Covariate

In Section 2, it is shown that a neural networks largely take roles as function approximators and, in fact, they themselves are a kind of recursive function. For example, consider a network that takes a 1 dimensional input, has 2 hidden layers, and outputs a 1 dimensional scalar value. Then the entire neural network can be written as:

$$\hat{y} = \underbrace{\sigma(w_3 \cdot [\underbrace{\sigma(w_2(\underbrace{\sigma(w_1x_1 + b_1)}_{z_1^{(1)}}) + b_2)}_{z_1^{(2)}}] + b_3)}_{z_1^{(3)} = \hat{y}} \tag{17}$$

Where $\sigma(.)$ is some activation function, $w = \{w_1, w_2\}$ is the vector of weights[14], and $\vec{b}$ is the set of biases. The functional form (and thus defining the *functional neural network*) of this model can be rewritten as:

$$\hat{y} = \underbrace{\sigma(w_3 \cdot [\underbrace{\sigma(w_2(\underbrace{\sigma(\int w_1(s)x_1(s)ds + b_1)}_{z_1^{(1)}}) + b_2)}_{z_1^{(2)}}] + b_3)}_{z_1^{(3)} = \hat{y}} \tag{18}$$

To make this more clear, let's take a step back in the process and consider the input neuron: x. In the usual case, this will be exactly a single observation with one co-variate. In the functional analogue here, this relates to a functional observation pre-defined as a linear combination of basis functions. Since the case(s) thus far have been limited to a scalar response, we can be content knowing that the

[14] In this 1 dimensional case

transformation of the functional observation $x(s)$ is a scalar as it exits out of the first hidden layer[15]. The rest of the network flows as in the usual case.

In order to move forward here, an integral approximation is required. The motivation for why we need this numerical approximation can be found in $z_1^{(1)}$ in (18). Examining it closer and expanding the basis from which it is formed, we find the following:

$$z_1^{(1)} = \sigma(\int w_1(s)x_1(s)ds + b_1) \tag{19}$$

$$= \sigma(\int \sum c_i\psi(s)x_1(s)ds + b_1) \tag{20}$$

$$= \sigma(\sum c_i \underbrace{\int \psi(s)x_1(s)ds}_{Approximated} + b_1) \tag{21}$$

The $\psi(s)$ vector is the evaluations of the basis functions of the coefficient function in the linear model along the continuum s. The $x(s)$ is the functional observation. Since it is likely that no closed form or simple integral of this form will exist, an integral approximation is required. The approximation technique of choice here is *Gaussian Quadrature*; this algorithm is specifically useful for polynomial functions [8] [5] and since we will be defining many of the basis with b-splines, this selection is optimal.

Now that we have a way to evaluate this integral, we can note that we have now transformed the problem from the one requiring the following derivatives:

$$\nabla R = \begin{pmatrix} \frac{\partial R}{\partial w_1(s)} & \frac{\partial R}{\partial w_2} & \frac{\partial R}{\partial w_3} & \frac{\partial R}{\partial b_1} & \frac{\partial R}{\partial b_2} & \frac{\partial R}{\partial b_3} \end{pmatrix}^T$$

To the following gradient:

$$\nabla R = \begin{pmatrix} \frac{\partial R}{\partial c_1} & \frac{\partial R}{\partial c_2} & \dots & \frac{\partial R}{\partial c_k} & \frac{\partial R}{\partial w_2} & \frac{\partial R}{\partial w_3} & \frac{\partial R}{\partial b_1} & \frac{\partial R}{\partial b_2} & \frac{\partial R}{\partial b_3} \end{pmatrix}^T$$

Where k is the number of basis functions used in the formulation of the coefficient function and R is the loss function. At this point, we have enough information to complete the forward pass. In the backward pass, the transformation made in (21) allows the gradient to be computed exactly as you would in a feed-forward neural network. The algorithm below describes the process and completes the definition of a FNN.

Algorithm 1 Backward Pass - Functional Feed-Forward Neural Network

Input: $\hat{y}, \theta, j.\ i = 0$
Output: Updated values for the set of parameters θ

1: While $j > i$:
2: Select a subset number, f of observations for which the neural network is run - a mini batch
3: Compute the gradient, ∇R for each observation and normalize
4: Take the average of the corresponding gradient values across all the observations
5: Add the average, weighted by some learning rate γ to the current values of the weights: $w_i = w_i + \frac{\partial R}{\partial_{w_i}} \cdot \gamma$ where w_i is some parameter of the network, $\hat{y}$
6: $i = i + 1$
7: Go back to 1

[15] That is, $z^{(1)}$ is a scalar

4.2 An Extension: Functional Residual Neural Networks

In the usual residual network case, the pass onto the next layer is accompanied by the value of the previous activation as well. For example, we had this equation:

$$z^{(1)} = \sigma(\vec{\alpha_{0_1}} + \vec{x}\boldsymbol{\alpha}^{(\mathbf{1})}) \tag{22}$$

But now, we will have this equation [7]:

$$z^{(1)} = \sigma(\vec{\alpha_{0_1}} + \vec{x}\boldsymbol{\alpha}^{(\mathbf{1})}) + \vec{x} \tag{23}$$

And, this is referred to as a residual because:

$$z^{(1)} - \vec{x} = \underbrace{\sigma(\vec{\alpha_{0_1}} + \vec{x}\boldsymbol{\alpha}^{(\mathbf{1})})}_{\text{residual}} \tag{24}$$

A residual neural network is characterized more specifically by residual blocks - that is, we have the property of adding back the previous activation value beginning and ending at any arbitrary point in the network. This is important to note because[16], as defined in Section 4.1, the activation values in the first layer, i.e. the inputs, are functions whereas the activation values deeper into the network are scalar values. The insight here is that if we begin to add residual blocks from the beginning, we will be passing a function into every subsequent layer whereas if we add residual blocks from anywhere after the first layer, the network will proceed exactly as would a residual neural network with regular inputs. To make this more clear, the former case will follow the structure in (18), where as the latter case, the *residual functional neural network* (ResFNN) will have the following structure:

$$z^{(1)} = \sigma(\int w_1(s)x_1(s)ds + b_1) + x(s) \tag{25}$$

Where $x(s)$ is now the *functional activation value* and hence, $z^{(1)}$ is now a function as opposed to a scalar. If the user wishes to proceed in the case as in (18), then forward pass continues exactly as you would expect. However, if the user wishes to proceed as in (25), then the network has an algorithmic shift.

In particular, the insight is that the entire network shifts to the case of going from input $\implies$ layer 1 in (18). This means that the numerical approximation of the integral needs to be computed in every neuron within every layer of the entire network resulting in a larger set of parameters to be estimated. For example, let's assume we had $k = 3$ as the number of basis coefficients describing every coefficient function. Then, considering the (nearly) canonical example of a network with a single neuron over two hidden layers, we observe that the number of coefficient parameters needing to be estimated increases from 3 to 9 - an $n \cdot k \cdot m$ increase where n is the number of layers for which we have the integral structure in the activation function, k being the number of coefficients defining the coefficient function in the linear model, and m being the number of neurons in each layer[17]. In Figure 4, a diagram is presented highlighting the possibilities.

4.3 On the Disintegration of the Neural Black-Box in the Functional Context

A key innovation of the functional neural network that is not available in the ordinary functional neural network is the move away from the "black-box" nature of these types of models. That is, since a leading contributor to the black-box label of neural networks is the inordinate amount of changing numbers, it can be illuminating to consider rather a function defined by these seemingly uninterpretable numbers. Remember that in the functional neural network, one of the (set of) weights we are estimating define a basis function. This basis function is extremely similar (in the 1 neuron

[16] In the functional case

[17] In this simple case, $m = 1, k = 3, n = 3$

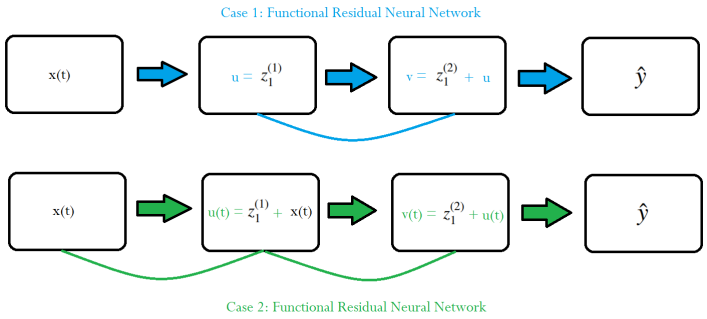

Figure 4: This is a figure illustrating the basic difference between the two types of functional residual neural networks. Case 1 is when the residual blocks begin after the first hidden layer. The 2^{nd} case is when the user wants the freedom to use the residual blocks anywhere in their network.

case) to the one predicted in the functional regression model. Essentially, the final set of weights here define the 11-basis β coefficient function and this can be compared with the usual one pulled from a function linear model.

For larger networks, we can see the movement of the functional coefficient in two directions: one across the iterations and the other across the depth of the network. To understand the former, Figure 5 may be illuminating:

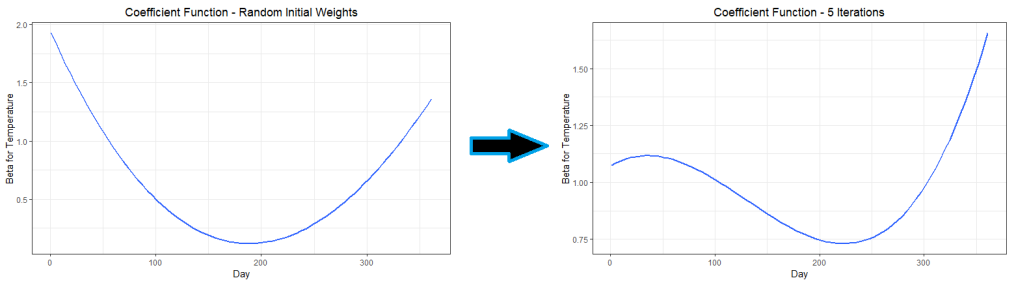

Figure 5: Illustration of how the functional neural coefficient changes over iterations.

We begin with some random weights which define the functional neural coefficient and, as the network runs through iterations, these weights are updated. The image on the left in Figure 5 is a curve produced by these initialized weights[18] whereas the curve on the right has adapted and shaped as the training data would indicate is best.

The latter is characterized more by the ability to see how these curves coexist - analogous to considering principal components. For example, if we had 3 hidden layers where the network is a ResFNN of the second case, then we could observe the movement of the coefficient curves for each of these layers across the iterations. The reconciliation of how these curves would work together is left at the discretion of the user; the author's advice would be to consider each curve as describing a different feature of the context which is being considered - for example, a question for weather data to ask could be: "what effects temperature over time, other than the seasons?". Alternatively, the aggregation of the curves may also be of value.

[18]The weights here being the coefficients of the basis functions

5 Results

5.1 Feed-Forward Functional Neural Network

In this section, an example is provided to show the predictive power of this algorithm. The data set used has information regarding the mean amount of precipitation in a year and the daily temperature for 35 Canadian cities. 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[19]. The results will be compared with prediction accuracy from a functional regression model and the usual neural network as defined in Section 2.

First, we define the functional observations. For this, a Fourier basis expansion was using with 65 basis functions defining each of the 35 cities [10]. The coefficient function of the regression model defining $z_1^{(1)}$ is defined using 11 basis coefficients. In total, there was 14[20] derivatives to be computed in this neural network per backward pass. There are also 11 integral approximations computed per forward pass. A various number of epochs were computed in an attempt to minimize overfitting and it seems that the choice of 10 full passes of the data set was optimal.

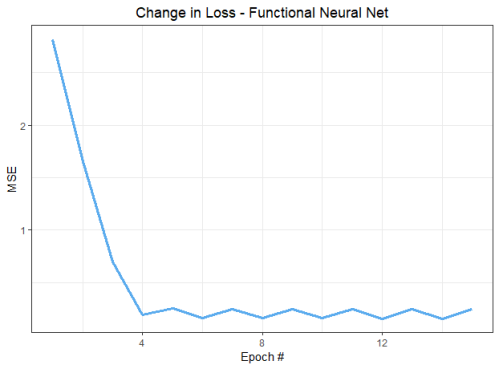

Figure 6: Loss plot using 15 epochs with a learning rate of 0.1

In Figure 6, notice the oscillating behaviour from epoch 4 to epoch 15. This behaviour is characterized by a high learning rate; the derivative update seems to make too big of a leap towards the local optimum and hence, never gets closer to the optimum (as the next update shifts the final output too much in the other direction) [9]. This problem was circumvented using an adaptive learning rate [6]; that is, the learning rate γ, adjusts as we move through the epochs to avoid the aforementioned behaviour.

The functional regression model was fit as would be expected and the mean squared error values for both of these fits is given in Table 1. The loss was considerably lower in the functional neural network when compared with the functional regression model.

Functional Regression	Functional Neural Network
0.484	0.0304

Table 1: Mean squared error values for the two different models

Predictions were made on the functional observation of *Resolute* to see how the models performed on a new observation, outside of the data set [1]. The results are given in Table 2.

[19] Note: there is only 1 covariate here and it is of a functional nature

[20] 11 from $w_1(t)$, and 1 each for w_2, b_1, and b_2

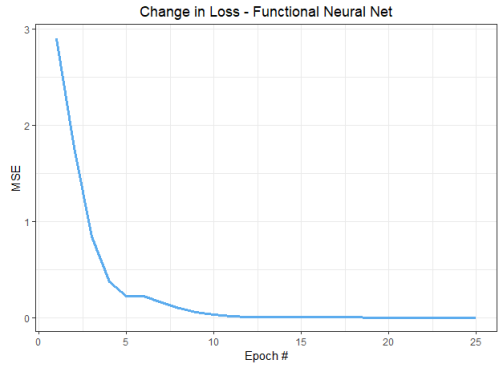

Figure 7: Loss plot using 25 epochs with an adaptive learning rate

Functional Regression	Functional Neural Network
0.242	6.25e-05

Table 2: Resolute prediction squared errors

Next, we can consider the y v. $\hat{y}$ and residual plots for the functional neural network given in Figure 8 and Figure 9. We aren't specifically concerned with the residual plot although it is nice given the interpretable nature of this functional neural network. The other plot shows a nearly perfect linear fit.

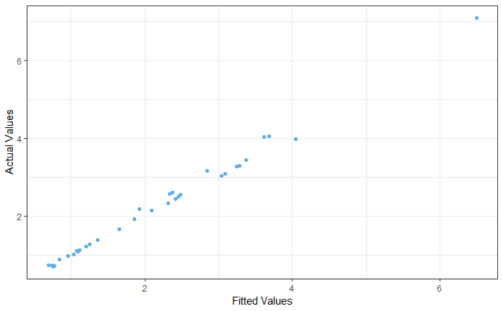

Figure 8: y vs. $\hat{y}$ plot for the functional neural network

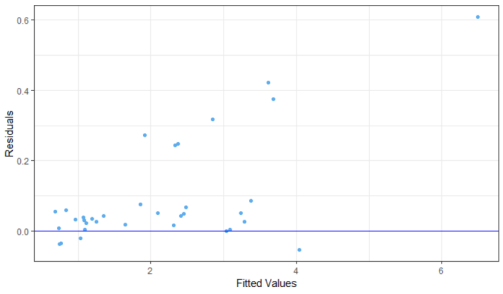

Figure 9: Residual plot for the functional neural network

Next, a comparison was made between a functional neural network and regular feed-forward neural network [3]. In the latter case, there were 365 covariates - each corresponding to the daily temperature for the given city. The same conditions were given as in the functional case and the functional neural network outperformed its multivariate counterpart[21]:

[21] Note, for some reason, the *neuralnet* package in **R** gives varying errors so in the markdown output, the value may be different. Regardless, no amount of this variability resulted in a lower error rate than the functional neural network

Neural Network	Functional Neural Network
0.173	0.0304

Table 3: Neural Network Comparison

5.2 The Functional Neural Coefficient

In Figure 10, the results from the functional linear regression of the coefficient is compared with the *functional neural coefficient*.

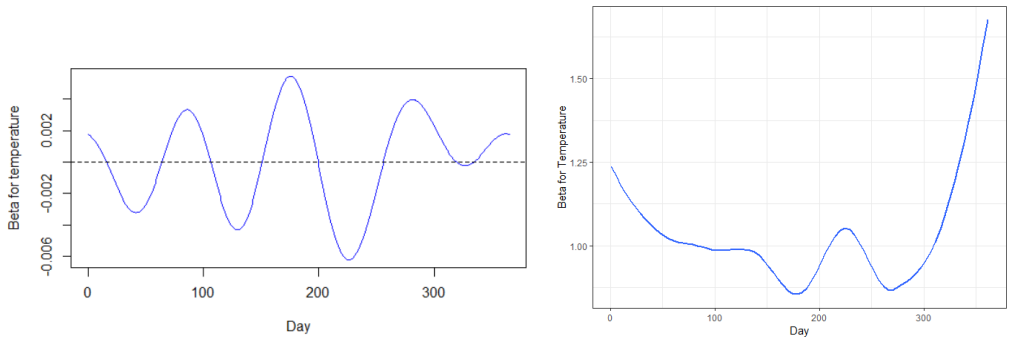

Figure 10: A comparison of the β coefficient functions. It seems that there is a difference here in that the functional neural network coefficient function states that there is an increased effect of temperature around the summer of summer on mean precipitation vs. the functional regression. Moreover, there seems to be more of a disparity between the effects over time in the functional neural network when compared with the more cyclical nature of the β coefficient function in the regression

6 Conclusions & Future Considerations

The advent and rise of neural networks has resulted in enormous breakthroughs in computer vision, classification, and scalar prediction. However, these advantages thus far had been limited to the multivariate space. This paper introduced one of a family of neural networks that extend into the functional case.

In particular, the neural networks introduced here encompass feed-forward and residual models. Algorithms were introduced that detailed the steps required to compute a solution for the network. These algorithms took advantage of Gaussian Quadrature and the usual backpropagation algorithm to compute integral approximations and weight updates for the functional observations (over the relevant time intervals). Moreover, an example was provided that showed that the functional neural network outperformed not only functional linear regression but also the classic neural networks with respect to squared error and test predictions. Lastly, it was shown that the usual black-box nature of regular neural networks is circumvented here; in fact, it is possible to extract out interpretable *functions* from these models akin to the coefficient functions in functional linear regressions.

To extend this project, algorithms will be developed for the other combination of input and output types. Additionally, a promising area of research moving forward could be an expansion into the new world of *Neural Ordinary Differential Equations* [4]. This approach will perhaps allow us to compute derivatives directly of the functions using the adjoint method which isn't a luxury in the discrete case(s) presented here. There is also potential for the application of classical statistical techniques used for uncertainty calculation; since we have the variation available to us on the β coefficient curves for each functional observation, it seems natural that we should be able to extend this to quantify the

uncertainty associated with these estimates. Regardless, this is an open area of research with lots of potential for creative extensions.

7 References

[1] user5594 1. *Predicting response from new curves using fda package in R*. URL: https://stats.stackexchange.com/questions/18470/predicting-response-from-new-curves-using-fda-package-in-r.

[2] 3Blue1Brown. *Backpropagation calculus | Deep learning, chapter 4*. Nov. 2017. URL: https://www.youtube.com/watch?v=tIeHLnjs5U8&index=4&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi.

[3] Guest Blog. *Creating and Visualizing Neural Network in R*. May 2018. URL: https://www.analyticsvidhya.com/blog/2017/09/creating-visualizing-neural-network-in-r/.

[4] Tian Qi Chen et al. "Neural Ordinary Differential Equations". In: *CoRR* abs/1806.07366 (2018). arXiv: 1806.07366. URL: http://arxiv.org/abs/1806.07366.

[5] John Cook. *Orthogonal Polynomials and Gaussian Quadrature*. URL: https://www.johndcook.com/OrthogonalPolynomials.pdf.

[6] *Gentle Introduction to the Adam Optimization Algorithm for Deep Learning*. Apr. 2019. URL: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/.

[7] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[8] *Is there any method better than Gauss Quadrature for numerical integration?* URL: https://www.researchgate.net/post/Is_there_any_method_better_than_Gauss_Quadrature_for_numerical_integration.

[9] Jeremy Jordan. *Setting the learning rate of your neural network.* Nov. 2018. URL: https://www.jeremyjordan.me/nn-learning-rate/.

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

[11] James O.. Ramsay and Bernard W. Silverman. *Functional data analysis*. Springer Science Business Media, 2010.

[12] Jerome Friedman Trevor Hastie Robert Tibshirani. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2009.

8 Acknowledgments

A special thank you to Jiguo Cao[22] and Kevin Multani[23] for their contributions to this work.

[22] Professor of Statistics, Simon Fraser University
[23] Department of Physics, Stanford