Neural Network Analysis of Psychological Data: A Step-by-Step Guide

Lingbo Tong and Zhiyong Zhang
University of Notre Dame

Abstract

Artificial Neural Networks (ANN) are receiving more and more attention in the field of psychology. With the availability of software programs, the wide application of ANN becomes possible. However, without a firm understanding of the basics of the ANN, issues can easily arise. This paper presents a step-by-step guide for implementing a feed-forward neural network (FNN) on a psychological dataset to illustrate the critical steps in building, estimating, and interpreting a neural network model. We start with a concrete example of a basic 3-layer FNN, illustrating the core concepts, the matrix representation, and the whole optimization process. By adjusting parameters and changing the model structure, we examine their effects on model performance. Then, we introduce accessible methods for interpreting model results and making inferences. Through the guide, we hope to help researchers avoid common problems in applying neural network models, and machine learning methods in general.

Keywords: Feed-forward Neural Network (FNN), Psychological Datasets

Neural networks, a cornerstone of artificial intelligence and machine learning, have experienced rapid development in recent years, becoming an indispensable tool across various research disciplines (LeCun et al., 2015). Their ability to learn from and make predictions or decisions based on data has been widely acknowledged and harnessed in multiple fields, ranging from downstream areas such as natural language processing and computer vision (Goldberg, 2022; Khan et al., 2018), to cross-cutting disciplines, e.g., computational neuroscience and bioinformatics (Min et al., 2017; Richards et al., 2019). In psychological research, deep learning has been applied to assisting psychiatrists and psychologists in various tasks such as mental disorder diagnosis (Iyortsuun et al., 2023; Su et al., 2020), suicide risk detection (Malhotra & Jindal, 2022; Tadesse et al., 2019), and personality assessment (**abdurahman2023deep**).

The exponential growth of the internet has facilitated the collection of massive datasets, combined with the advent of computing resources, granting scientists opportunities to create and distribute large-scale deep learning models (Abadi et al., 2016; Kaddour et al., 2023). However, the current research landscape seems oriented towards these large-scale models and extensive datasets, leaving a gap in exploring the efficacy of portable models in smaller, lab-collected datasets, standard in social sciences. This issue is particularly pronounced within the field of psychology, where researchers often struggle to adopt neural-network-based methods in their own research. Therefore, a tutorial featuring clear and concrete examples focusing on these approaches will be beneficial,

as it could assist psychologists in comprehending the fundamental mechanism of neural network models and their usage.

Addressing this knowledge gap, the present paper offers a step by step tutorial for implementing neural networks in psychological research, through the analysis of the data from the Advanced Cognitive Training for Independent and Vital Elderly (ACTIVE) study (Jobe et al., 2001). Particularly, we show how to apply a fundamental 3-layer feed-forward neural network (FNN) to the dataset and explain the neuron update processes involved. Subsequently, we discuss model optimization, examining how various parameter adjustments and model structures can influence the network's performance. Additionally, we explore methods for interpreting model results and making inferences. Finally, we conclude with an overview on the development of deep learning models in recent years, offering a primer for psychologists aiming to incorporate neural networks into their research. The tutorial should provide researchers with a clear understanding on how the methods work and can help researchers in conditions ranging from building their own models to applying existing techniques such as ChatGPT.

Dataset

This paper's data are drawn from the ACTIVE (Advanced Cognitive Training for Independent and Vital Elderly) dataset (Jobe et al., 2001). Originating from a large-scale study conducted from 1999 to 2001 in the United States, the ACTIVE dataset involved six field sites and aimed to evaluate the impact of three cognitive interventions (focused on memory, executive reasoning, and processing speed) on daily cognitive activities in older adults. The study included 2,832 participants, with assessments conducted at baseline, post-training, and annually thereafter. A subset of 11 variables from the ACTIVE study is presented in Table 1, which will be used in our illustration. After excluding records with empty or invalid values, the dataset comprises 1,573 participants.[1]

Table 1

Variable names and descriptions in the ACTIVE dataset.

Variable	Meaning	Occasion
age	Participant age	Baseline
edu	Years of education	Baseline
sex	1: male; 2: Female	Baseline
booster	Whether received booster training	Baseline
reason	Reasoning ability	Baseline
ufov	Useful field of view variable	Baseline
mmse	Mini-mental state examination total score	Baseline
hvltt	Hopkins Verbal Learning Test total score at time 1	Baseline
hvltt2	Hopkins Verbal Learning Test total score at time 2	Post-training
hvltt3	Hopkins Verbal Learning Test total score at time 3	One year later
hvltt4	Hopkins Verbal Learning Test total score at time 4	Two years later

[1] The dataset and the code are available at https://github.com/Stan7s/Neural-Network-Tutorial

Feed-Forward Neural Network

A feed-forward neural network (FNN), also known as a multilayer perceptron (MLP), is a fundamental artificial neural network used in machine learning and deep learning (Schmidhuber, 2015). It represents a basic building block for more complex neural network architectures. In an FNN, information flows in one direction, from the input layer through one or more hidden layers to the output layer, without any feedback loops (Sazli, 2006). Each layer consists of interconnected neurons that process and transform the input information from the previous layer.

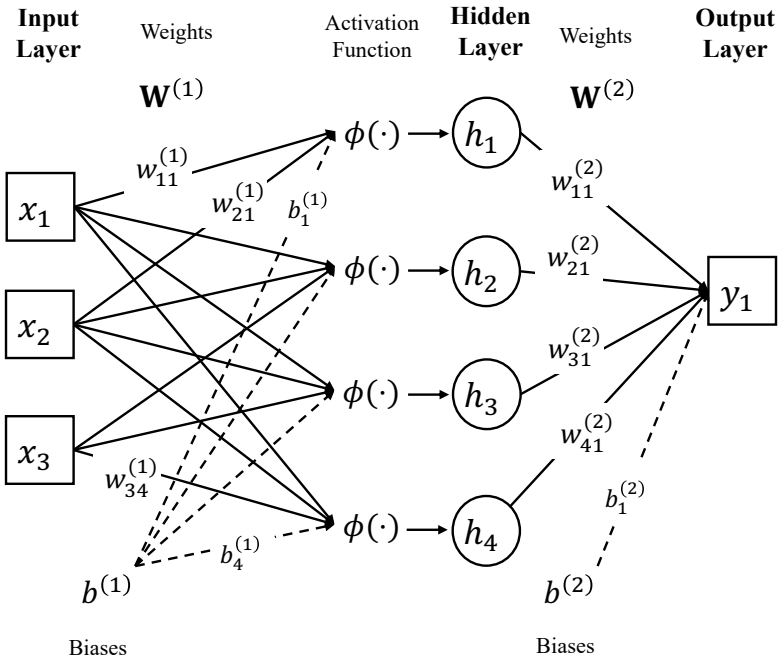
The simplest FNN contains three layers: an input layer, a hidden layer, and an output layer, as shown in Figure 1. Formally, the model can be defined as

$$\hat{\mathbf{Y}} = \phi(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}) \cdot \mathbf{W}^{(2)} + \mathbf{b}^{(2)}. \tag{1}$$

Here, $\mathbf{X}$ denotes the input data, $\mathbf{W}^{(1)}$ denotes the weight parameters connecting the input layer and the hidden layer, and $\mathbf{b}^{(1)}$ denotes the bias parameters associated with the hidden layer. Similarly, $\mathbf{W}^{(2)}$ and $\mathbf{b}^{(2)}$ denote the weights and biases between the hidden layer and the output layer, respectively. $\phi(\cdot)$ represents a linear or nonlinear activation function, and $\hat{\mathbf{Y}}$ indicates the predicted outcomes of the model in the output layer. The actual outcomes are represented by $\mathbf{Y}$, observed in the data. We now explain each component using the ACTIVE data example in subsequent subsections. Suppose in the example, we want to predict the verbal test score at time 4 (hvltt4) using age, years of education, and gender.

Figure 1

A three-layer feed-forward neural network (FNN).



Input Layer

The input layer of the FNN is represented as an $n \times p$ matrix, with n denoting the number of participants and p signifying the number of input variables or predictors. As an example, we consider three predictors ($p = 3$): *age*, *years of education*, and *sex*. For illustration, we let $n = 5$ here. The resulting input matrix appears as follows with the collected data in the ACTIVE study:

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \\ x_{41} & x_{42} & x_{43} \\ x_{51} & x_{52} & x_{53} \end{bmatrix} = \begin{bmatrix} 65 & 12 & 1 \\ 70 & 13 & 0 \\ 83 & 12 & 1 \\ 72 & 16 & 0 \\ 66 & 13 & 1 \end{bmatrix}. \tag{2}$$

Here, x_{ij} indicates the value for the i^{th} participant of the j^{th} predictor. For instance, the first row in the input matrix represents a record from a 76-year-old female participant with 12 years of education. The data in the input layer are known and observed.

A common preprocessing step is to normalize the input features by subtracting the mean and dividing by the standard deviation, i.e., standardizing the data to have zero mean and unit variance (LeCun et al., 2002). This technique ensures that all input features are on a similar scale, preventing any single feature from dominating the learning process due to its larger values (Zheng & Casari, 2018). Furthermore, it helps gradient-based optimization algorithms converge faster and more effectively during training (Goodfellow et al., 2016). After normalization, the input data becomes:

$$\mathbf{X} = \begin{bmatrix} -1.4476 & -0.6565 & 0.5421 \\ -0.548 & -0.2731 & -1.8447 \\ 1.7909 & -0.6565 & 0.5421 \\ -0.1882 & 0.8772 & -1.8447 \\ -1.2677 & -0.2731 & 0.5421 \end{bmatrix}. \tag{3}$$

Hidden Layer

A hidden layer in an FNN consists of a set of neurons between the input layer and the output layer. Each neuron in a hidden layer transforms the values from the previous layer with a weighted linear summation followed by a linear or nonlinear activation function.

The layer transformation takes the form $\mathbf{H} = \phi(\mathbf{X}\mathbf{W} + \mathbf{b})$, where $\mathbf{W}$ denotes the weight matrix that represents the linear transformation of the input from the previous layer to the current layer, $\mathbf{b}$ is the bias vector that adjusts neurons' activation thresholds, $\phi(\cdot)$ is the activation function, and $\mathbf{H}$ is the outcome of the layer transformation. Note that the bias vector $\mathbf{b}$ is added to each row of the product matrix $\mathbf{X}\mathbf{W}$ —a procedure called *broadcasting*. This is the same as $\mathbf{X}\mathbf{W} + \mathbf{1} \cdot \mathbf{b}$ with $\mathbf{1}$ denoting a column vector of ones in strict matrix algebra notations.

The dimensions of the weight matrix connecting the input layer to the hidden layer are $p \times h$, and these of the biases are $1 \times h$, respectively, where h denotes the number of neurons in the hidden layer. For instance, if the hidden layer comprises 4 neurons, the weights and biases would be structured as follows:

$$\mathbf{W}^{(1)} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} & w_{14}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} & w_{24}^{(1)} \\ w_{31}^{(1)} & w_{32}^{(1)} & w_{33}^{(1)} & w_{34}^{(1)} \end{bmatrix} \quad \mathbf{b}^{(1)} = \begin{bmatrix} b_1^{(1)} & b_2^{(1)} & b_3^{(1)} & b_4^{(1)} \end{bmatrix}, \tag{4}$$

where $w_{ij}^{(1)}$ denotes the i^{th} weight value for the j^{th} neuron of the hidden layer, and $b_j^{(1)}$ denotes the bias associated with the j^{th} neuron of that layer. During the linear summation, the input data matrix $\mathbf{X}$ is multiplied by the weight matrix $\mathbf{W}$, yielding a matrix of dimensions $n \times h$. Subsequently, the bias vector $\mathbf{b}$ is added to each row of the product matrix through broadcasting.

The linear summation is typically followed by a nonlinear activation function, although a linear function can also be used, enabling the network to capture complex patterns and relationships within the data. The activation function, denoted as $\phi(\cdot)$, will be applied to each element in the output matrix of the linear transformation.

Different nonlinear functions could be used as activation functions in FNN. One widely applied function is the Sigmoid function, characterized by its S-shaped curve, transforming input values into a value between 0 and 1. The function is represented as:

$$\text{Sigmoid}(t) = \frac{1}{1 + e^{-t}}. \tag{5}$$

Another popular activation function is the ReLU function, a piecewise linear function that outputs the input value if it's positive and zero if it's negative. Formally,

$$\text{ReLU}(t) = \max(0, t). \tag{6}$$

ReLU has become a favored choice in FNNs due to its computational efficiency and ability to mitigate the vanishing gradient problem during training (Agarap, 2018; Yu & Zhu, 2020). In this paper, we will consistently utilize ReLU in all our experiments.

Following the activation function, we obtain the output matrix $\mathbf{H}$, maintaining dimensions $n \times h$, representing the values post-transformation of the input data.

Output Layer

The output layer transforms the values derived from the last hidden layer, yielding the final output. For regression models where the output can be any continuous value, it is common not to have an activation function, or to use an identity activation function, in the output layer. Therefore, this transformation is mathematically represented as $\mathbf{Y} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$. Here, the weight matrix $\mathbf{W}^{(2)}$ and the bias vector $\mathbf{b}^{(2)}$ have dimensions of $h \times q$ and $1 \times q$, respectively, with q denoting the number of output variables. The number of output variables depends on the structure of observed data and research questions, i.e., how many outcome variables to predict. In the context of this example, with the single variable *hvltt4* specified as the output, $q = 1$, determining the weights and biases as follows:

$$\mathbf{W}^{(2)} = \begin{bmatrix} w_{11}^{(2)} \\ w_{21}^{(2)} \\ w_{31}^{(2)} \\ w_{41}^{(2)} \end{bmatrix} \quad \mathbf{b}^{(2)} = \begin{bmatrix} b_1^{(2)} \end{bmatrix}. \tag{7}$$

Consequently, the final output matrix $\hat{\mathbf{Y}}$ possesses the dimensions of $n \times q$. Each element y_{ij} within $\hat{\mathbf{Y}}$ signifies the predicted outcome for the i^{th} participant concerning the j^{th} item. In instances where $q = 1$, the final output would be:

$$\hat{\mathbf{Y}} = \begin{bmatrix} \hat{y}_{11} \\ \hat{y}_{21} \\ \hat{y}_{31} \\ \hat{y}_{41} \\ \hat{y}_{51} \end{bmatrix}. \tag{8}$$

Considering all the weight matrices and bias vectors combined, the total number of parameters to estimate in a three-layer FNN is given by the formula

$$\text{\# params} = (p + 1) \times h + (h + 1) \times q. \tag{9}$$

For our specific case, this calculation becomes $(3 + 1) \times 4 + (4 + 1) \times 1 = 21$ parameters. The process of updating these parameters until the model reaches optimization will be described in the following section.

Model Training and Evaluation

In machine learning, model training refers to the process where a model learns from a dataset by adjusting its parameters to minimize errors, and model evaluation refers to the assessment of how well a model's predictions align with actual outcomes.

Data Split

Before starting model training on our dataset, we need to set aside a portion of the dataset for evaluation. A common practice is to divide the original dataset into different subsets for training, validation (optional) and testing. The training set is used for the learning process. The validation set helps to fine-tune the hyperparameters, determine the appropriate time to stop the training process, and facilitate model selection. The test set is used for final evaluation.

In this tutorial, we employ a two-tiered splitting strategy: Initially, we randomly selected 15% of the entire dataset as the test set. Subsequently, from the remaining data, we randomly take 15% as the validation set. This partitioning results in training, validation, and test sets comprising approximately 72.25%, 12.25%, and 15% of the original dataset, respectively.

Training

Training an FNN is the process of finding the weights and biases in a neural network. It involves several key stages. The process begins with *initialization*, where the network's parameters (weights and biases) are assigned selected, often random, initial values. For example, for each linear layer with a input features, its weights and biases can be initialized from $U(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{a}$, through random number generation. In the context of our example, the initialized parameter matrices appear as follows:

$$\mathbf{W}^{(1)} = \begin{bmatrix} 0.0387 & -0.9131 & 0.0134 & -0.4025 \\ -0.8408 & -0.7913 & 0.0134 & 0.0391 \\ 0.3027 & 0.7905 & 0.7644 & 0.1751 \end{bmatrix} \quad \mathbf{b}^{(1)} = \begin{bmatrix} 0.4067 & 0.5735 & -0.3693 & -0.0427 \end{bmatrix} \tag{10}$$

$$\mathbf{W}^{(2)} = \begin{bmatrix} -1.0246 \\ -0.7404 \\ 0.5110 \\ -0.0512 \end{bmatrix} \quad \mathbf{b}^{(2)} = \begin{bmatrix} -0.0290 \end{bmatrix}. \tag{11}$$

During *forward propagation*, input data traverse the network, producing predictions. First, the input data (the example data in Eq 2) goes through the linear summation in the hidden layer, which leads to:

$$\mathbf{Z} = \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)} = \begin{bmatrix} 1.0667 & 2.8433 & 0.0169 & 0.6092 \\ 0.0567 & -0.1683 & -1.7904 & -0.1558 \\ 1.1921 & -0.1138 & 0.0603 & -0.6943 \\ -0.8966 & -1.4071 & -1.7702 & -0.2557 \\ 0.7513 & 2.3756 & 0.0244 & 0.5518 \end{bmatrix}. \tag{12}$$

Then, the activation function is applied, here a ReLU, to get:

$$\mathbf{H} = \text{ReLU}(\mathbf{Z}) = \begin{bmatrix} 1.0667 & 2.8433 & 0.0169 & 0.6092 \\ 0.0567 & 0 & 0 & 0 \\ 1.1921 & 0 & 0.0603 & 0 \\ 0 & 0 & 0 & 0 \\ 0.7513 & 2.3756 & 0.0244 & 0.5518 \end{bmatrix}. \tag{13}$$

And finally, the output layer can be obtained as:

$$\hat{\mathbf{Y}} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \begin{bmatrix} -3.2497 \\ -0.0871 \\ -1.2196 \\ -0.0290 \\ -2.5735 \end{bmatrix}. \tag{14}$$

Now, we have finished the first round of forward propagation, resulting in a set of predictions $\hat{\mathbf{Y}}$. The difference between these predictions and actual values is quantified by a *loss function*, reflecting how well the network is performing at the current stage in the training process. One of the most popular loss functions for continuous data is the mean squared error (MSE) (Gareth et al., 2013), which can be defined as:

$$\text{MSE} = \frac{1}{nq}\sum_{i=1}^{n}\sum_{j=1}^{q}(y_{ij} - \hat{y}_{ij})^2 \tag{15}$$

where n is the sample size, y_{ij} is the observed value of the j^{th} dependent variable for the i^{th} participant, and $\hat{y}_{ij}$ is the corresponding predicted value generated from the neural network.

Given the observed values of the outcome variable

$$\mathbf{Y} = \begin{bmatrix} 34 \\ 27 \\ 27 \\ 26 \\ 27 \end{bmatrix}, \tag{16}$$

and the prediction in the output layer in Eq (14), the MSE is then calculated as

$$\text{MSE} = \frac{1}{5}\sum_{i=1}^{5}(y_{i1} - \hat{y}_{i1})^2 = 178.79, \tag{17}$$

which is the average squared difference between the predicted values and the actual values in the dataset. Taking the square root of this value, we obtain a Root Mean Square Error (RMSE) of 13.37. This metric quantifies how far the model's predictions are from the observed values on average after the first round of forward propagation.

The objective of training the neural network is to minimize the loss value by obtaining the "best" values for the weights and biases. This is achieved through *backpropagation*, where the network adjusts its weights and biases based on this loss. The stochastic gradient descent (SGD) algorithm is commonly employed to optimize these adjustments to get an updated set of parameter values. Mathematically, the update rule for SGD is given by:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \cdot \frac{\partial l}{\partial \boldsymbol{\theta}} \tag{18}$$

where $\boldsymbol{\theta}$ represents a vector of all the model's parameters, t is the iteration index, η is the learning rate — a positive scalar determining the step size in the direction opposite to the gradient, and $\frac{\partial l}{\partial \boldsymbol{\theta}_t}$ is the gradient of the model loss l with respect to $\boldsymbol{\theta}$.

Getting the gradient is critical. Let's derive the gradients step by step using the example data, starting from the MSE model loss function:

$$l = \text{MSE}^* = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{q}(y_{ij} - \hat{y}_{ij})^2. \tag{19}$$

Here, l is a scalar denoting the average squared loss. Note that we have modified the standard MSE loss to MSE* for computational ease during differentiation. Firstly, we omit the constant coefficient $\frac{1}{nq}$, since both n (the number of samples) and q (the number of output variables per sample) are constants and do not affect the optimization process. Secondly, we introduce a factor of $1/2$. This adjustment is made because the squared term in MSE leads to a factor of 2 when we take its derivative. By incorporating the $1/2$ factor, the 2 will be canceled out when differentiating, simplifying the equations.

The partial derivative of the loss function with respect to each predicted value $\hat{y}_{ij}$ is:

$$\frac{\partial l}{\partial \hat{y}_{ij}} = \frac{\partial}{\partial \hat{y}_{ij}}\left(\frac{1}{2}\sum_{k=1}^{n}\sum_{l=1}^{q}(y_{kl} - \hat{y}_{kl})^2\right) = \hat{y}_{ij} - y_{ij}. \tag{20}$$

In matrix form, this gradient is a matrix of the same dimensions as $\hat{\mathbf{Y}}$, where each element is the derivative with respect to the corresponding element in $\hat{\mathbf{Y}}$. Thus, it can also be denoted as:

$$\frac{\partial l}{\partial \hat{\mathbf{Y}}} = \hat{\mathbf{Y}} - \mathbf{Y} = \begin{bmatrix} -37.2497 \\ -27.0871 \\ -28.2196 \\ -26.0290 \\ -29.5735 \end{bmatrix}. \tag{21}$$

The subsequent steps of computing the gradients are guided by the chain rule of derivatives, formally, $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x}$. When dealing with functions in matrix formats, and considering that z is a scalar while $\mathbf{X}$ and $\mathbf{Y}$ are matrices, we can derive the following chain rule:

$$z = f(\mathbf{Y}), \mathbf{Y} = \mathbf{X}\mathbf{W} + \mathbf{B} \rightarrow \frac{\partial z}{\partial \mathbf{X}} = \frac{\partial z}{\partial \mathbf{Y}}\mathbf{W}^T, \ \frac{\partial z}{\partial \mathbf{W}} = \mathbf{X}^T\frac{\partial z}{\partial \mathbf{Y}}. \tag{22}$$

This formulation becomes especially handy when deriving gradients in deep learning since our loss function typically outputs a scalar value, while the model parameters are in matrix or vector form. For instance, since $\mathbf{Y} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$, the gradient of the loss l with respect to the weight matrix $\mathbf{W}^{(2)}$ is:

$$\frac{\partial l}{\partial \mathbf{W}^{(2)}} = \mathbf{H}^T\frac{\partial l}{\partial \hat{\mathbf{Y}}} = \mathbf{H}^T(\hat{\mathbf{Y}} - \mathbf{Y}) = \begin{bmatrix} -97.13 \\ -176.1658 \\ -3.0513 \\ -39.0105 \end{bmatrix}. \tag{23}$$

Likewise, the gradient of the loss l with respect to the hidden layer output $\mathbf{H}$ is:

$$\frac{\partial l}{\partial \mathbf{H}} = \frac{\partial l}{\partial \hat{\mathbf{Y}}}(\mathbf{W}^{(2)})^T = (\hat{\mathbf{Y}} - \mathbf{Y})(\mathbf{W}^{(2)})^T = \begin{bmatrix} 38.1661 & 27.5797 & -19.0346 & 1.9072 \\ 27.7534 & 20.0553 & -13.8415 & 1.3869 \\ 28.9138 & 20.8938 & -14.4202 & 1.4448 \\ 26.6693 & 19.2719 & -13.3008 & 1.3327 \\ 30.301 & 21.8962 & -15.112 & 1.5142 \end{bmatrix}. \quad (24)$$

The gradient of l with respect to $\mathbf{b}^{(2)}$ can be derived in the same way. Since $\mathbf{b}^{(2)}$ is actually $\mathbf{1} \cdot \mathbf{b}^{(2)}$, with 1 denoting a column vector of ones in strict matrix algebra notations, the gradient concerning $\mathbf{b}^{(2)}$ is given by:

$$\frac{\partial l}{\partial \mathbf{b}^{(2)}} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}_n^T \frac{\partial l}{\partial \hat{\mathbf{Y}}} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}_n^T (\hat{\mathbf{Y}} - \mathbf{Y}) = \begin{bmatrix} -148.1589 \end{bmatrix}. \quad (25)$$

Next, we come to the output of the first linear transformation $\mathbf{Z}$. Given $\mathbf{H} = \text{ReLU}(\mathbf{Z})$, and the derivative of $u = \text{ReLU}(t)$:

$$\frac{\delta u}{\delta t} = \text{sign}(\max(t, 0)) = \begin{cases} 1 & t > 0 \\ 0 & t \leq 0 \end{cases}, \quad (26)$$

we can apply the chain rule to get:

$$\frac{\partial l}{\partial \mathbf{Z}} = \frac{\partial l}{\partial \mathbf{H}} \cdot \frac{\partial \mathbf{H}}{\partial \mathbf{Z}} = (\hat{\mathbf{Y}} - \mathbf{Y})(\mathbf{W}^{(2)})^T \cdot \text{sign}(\max(\mathbf{Z}, 0)). \quad (27)$$

Then, by $\mathbf{Z} = \mathbf{H}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}$, we get

$$\frac{\partial l}{\partial \mathbf{W}^{(1)}} = \mathbf{H}^T \frac{\partial l}{\partial \mathbf{Z}} = \mathbf{H}^T (\hat{\mathbf{Y}} - \mathbf{Y})(\mathbf{W}^{(2)})^T \cdot \text{sign}(\max(\mathbf{Z}, 0)), \quad (28)$$

and

$$\frac{\partial l}{\partial \mathbf{b}^{(1)}} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}_n^T \frac{\partial l}{\partial \mathbf{Z}} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}_n^T (\hat{\mathbf{Y}} - \mathbf{Y})(\mathbf{W}^{(2)})^T \cdot \text{sign}(\max(\mathbf{Z}, 0)). \quad (29)$$

Plugging in numerical values, we have:

$$\frac{\partial l}{\partial \mathbf{Z}} = \begin{bmatrix} 38.1661 & 27.5797 & -19.0346 & 1.9072 \\ 27.7534 & 0 & -0 & 0 \\ 28.9138 & 0 & -14.4202 & 0 \\ 0 & 0 & -0 & 0 \\ 30.301 & 21.8962 & -15.112 & 1.5142 \end{bmatrix}, \quad (30)$$

$$\frac{\partial l}{\partial \mathbf{W}^{(1)}} = \begin{bmatrix} -60.3886 & -42.3639 & 30.7332 & -2.6078 \\ -35.5585 & -24.945 & 18.0965 & -1.5355 \\ -46.7778 & -32.8156 & 23.8063 & -2.02 \end{bmatrix}, \tag{31}$$

and

$$\frac{\partial l}{\partial \mathbf{b}^{(1)}} = \begin{bmatrix} 148.336 & 104.0608 & -75.4916 & 6.4057 \end{bmatrix}. \tag{32}$$

Now that we have derived gradients for all model parameters and hidden layer outputs, we can update the model parameters following (18) to be:

$$\mathbf{W}^{(1)}_{t=1} = \mathbf{W}^{(1)}_{t=0} - \eta * \frac{\partial l}{\partial \mathbf{W}^{(1)}} \tag{33}$$

$$\mathbf{b}^{(1)}_{t=1} = \mathbf{b}^{(1)}_{t=0} - \eta * \frac{\partial l}{\partial \mathbf{b}^{(1)}} \tag{34}$$

$$\mathbf{W}^{(2)}_{t=1} = \mathbf{W}^{(2)}_{t=0} - \eta * \frac{\partial l}{\partial \mathbf{W}^{(2)}} \tag{35}$$

$$\mathbf{b}^{(2)}_{t=1} = \mathbf{b}^{(2)}_{t=0} - \eta * \frac{\partial l}{\partial \mathbf{b}^{(2)}}. \tag{36}$$

For example, when $\eta = 0.0001$, $\mathbf{W}^{(1)}_{t=1}$ becomes

$$\begin{bmatrix} 0.0449 & -0.9086 & 0.0103 & -0.4022 \\ -0.8372 & -0.7887 & 0.0116 & 0.0393 \\ 0.3075 & 0.7939 & 0.762 & 0.1753 \end{bmatrix}. \tag{37}$$

This concludes the first round of forward and backward propagation. The process can be repeated. Concrete values of gradients and updated parameters during the first three rounds and the associated Python code for reproducing these values can be found in the code repository.

The 5 samples or participants used in the example are called one *batch* of data, many times also referred to as *minibatch* (Masters & Luschi, 2018). In neural network training, a batch is a subset of the training dataset used for a single step of gradient calculation and weight updating. The number of training samples in one batch is called *batch size*. The neural network updates its parameters with each batch processed, and thus the batch size can influence both the duration of training and the performance of the model. Once the network has processed all the data in the training dataset, batch by batch, this completes one training iteration, typically called an *epoch*.

The training process typically involves multiple epochs. Throughout these epochs, the loss should ideally decrease, signifying that the network is learning the data. Training usually continues until the loss reaches an acceptable level or stops decreasing significantly, indicating that the model has converged.

Both training and validation sets are involved in the training process. Specifically, after each training epoch, we test the model on the validation set without updating the parameters. This approach gives us insight into how well the model is learning over time. For instance, we can

terminate the training if the model's loss on the validation set stops decreasing (or decreases by less than a tiny amount) over a number of consecutive epochs such as 20. This technique, known as *early stopping*, prevents overfitting by stopping the training before the total number of epochs reaches a pre-defined maximum, which is typically set to a large value. Figure 2 illustrates how the FNN's training and validation loss changed while being trained with a learning rate of 0.0001 in SGD and a batch size of 64. The training loss curve appears smooth because we performed gradient descent on the training set that has more participants, while the validation loss curve has more fluctuations. The curve flattened out around the 80th training epoch, and was terminated around the 100th epoch, where early stopping was triggered.

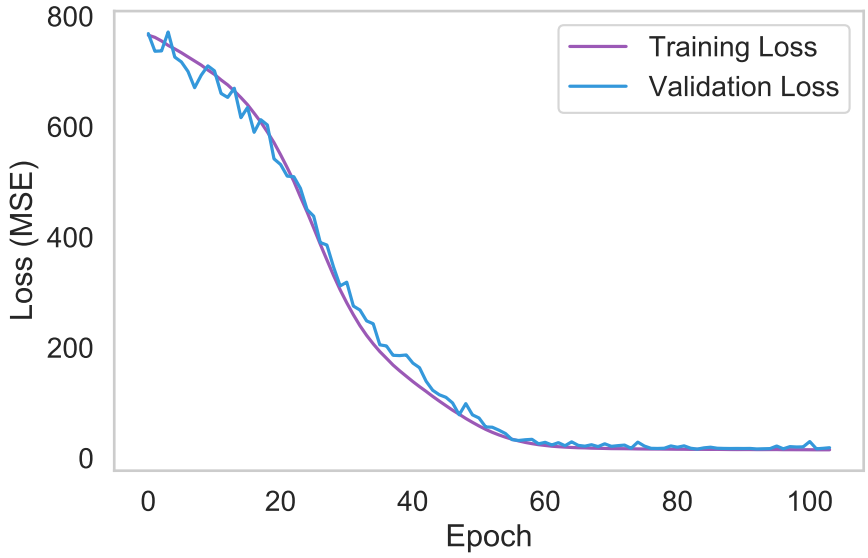


Figure 2

Change of MSE loss with the number of epochs during the training stage.

Evaluation

After training the FNN, our next step is to evaluate its performance on the test set. Depending on the task and focus, different metrics can be used to evaluate the performance. In this paper, we continue to use MSE to assess the performance of model predictions. Other popular metrics include Mean Absolute Error (MAE), Root Mean Square Error (RMSE), and R^2 for regression (Hyndman & Koehler, 2006; Miles, 2005), and Accuracy, Precision, Recall, and F1-score for classification (Powers, 2020).

One primary concern during evaluation is to check the model's *generalizability*. Specifically, *overfitting* refers to a model that performs exceptionally well on training data, but poorly on unseen data. Conversely, *underfitting* occurs when the model is too simple to capture underlying data patterns, resulting in suboptimal performance on both training and testing data (Goodfellow et al., 2016). In our analyses using the ACTIVE dataset, overfitting can occur if we develop an overly complex neural network model. In this case, the model may have almost perfect predictive accuracy

on the training data, suggesting that it essentially memorizes the dataset, including the noise and outliers, rather than learning the underlying patterns. However, when this model is tested on a test set, its accuracy can drop dramatically because its learned patterns do not generalize well beyond the training data. On the other hand, an underfitted model may be due to the use of overly simple methods, such as linear regression that predicts the same scores using only one or two predictors (e.g., age and sex). Such a model may perform equally poorly on both the training and test sets because it is too simple to capture the complex relationship between input and outcome variables.

Optimizing Model Performance

The performance of a neural network is associated with a variety of elements. Optimizing these elements can be crucial for maximizing prediction accuracy. Key factors include hyperparameters like the learning rate of SGD and the size of training batches, as well as architectural aspects such as the number of neurons and layers. This section is designed to demonstrate the impact of each factor on the model's performance, offering practical insights for researchers looking to enhance their neural network models.

We base our exploration on the FNN illustrated in Figure 2. This model, with a single hidden layer of 4 neurons, was initially trained with an SGD learning rate of 0.0001 and a batch size of 64. In the following subsections, we will methodically adjust several elements – learning rate, batch size, number of neurons, and number of layers – and examine their influence on the model's performance. At the end, we introduce grid search, an approach for identifying the most suitable hyperparameters and network architecture for a given dataset.

Adjusting Learning Rate

The learning rate in SGD controls the step size of parameter updates towards minimizing the loss function. A learning rate that is too small can lead to premature convergence to the local minimum. Conversely, a too big learning rate might cause the model to oscillate without stabilizing, prolonging the convergence time and also increasing the chance of reaching suboptimal solutions. The optimal selection of learning rates is crucial for efficient and effective model training.

In our experiment, we varied the learning rate across a spectrum from 0.1 to 1e-5. To ensure the reliability of the result, we trained 100 distinct FNN models for each learning rate value (i.e., 100 replications under each condition) using the same training set. (To reduce randomness) Their performance was evaluated on the validation set, applying an early stopping strategy to halt training if the validation loss decreased by 0.0001 or less over 20 consecutive epochs. Note that some models did not converge due to issues such as vanishing or exploding gradients. We remove these models and will discuss this matter in later subsections.

The compiled results are presented in Figure 3, showcasing a box plot that depicts the distribution of MSE across converged replications at each learning rate setting. We noted an initial reduction in MSE as the learning rate increased from 1e-5 to 0.001. However, a subsequent rise in the mean and variance of MSE was observed when the learning rate was further increased from 0.01 to 0.1, aligned with the mechanism shown in Figure 3.

Changing Batch Size

The value of batch size is another hyperparameter in neural network training, profoundly impacting model performance. Larger batch sizes typically offer a more precise gradient estimation,

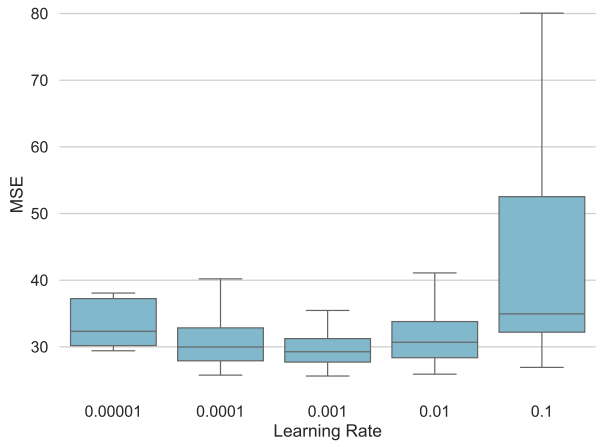


Figure 3

Model performance with different SGD learning rates.

promoting stable and consistent updates during gradient descent. However, they would require more computing resources and carry the risk of converging to local minima, potentially inhibiting optimal performance.

Conversely, smaller batch sizes introduce a level of noise that acts as a regularizer (Wilson & Martinez, 2003), often resulting in a reduced generalization error and a swifter and more robust convergence path (Masters & Luschi, 2018). Smaller batch sizes also align well with environments with memory constraints, such as GPU-based training. However, training with smaller batches typically necessitates a greater number of training steps and may also require a lower learning rate to preserve stability, given the higher variance in gradient estimation. These factors can potentially increase the total runtime of the training process (Goodfellow et al., 2016).

In practical applications, smaller batch sizes are generally favored. Bengio (2012) recommended a batch size range from 1 to a few hundred for effective training, with 32 as a common default. Research by Masters and Luschi (2018) indicates optimal test performance even with batch sizes as small as 2. Nonetheless, the ideal batch size is contingent on factors like the neural network's architecture and the dataset's characteristics (Radiuk, 2017).

The base FNN model depicted in Figure 2 used a batch size of 64. To explore the impact of different batch sizes, we conducted experiments with batch sizes of 8, 16, 32, 64, and 128, while keeping other parameters constant. Consistent with our approach for learning rate adjustment, each batch size setting underwent 100 replications. The outcomes on the validation set are illustrated in Figure 4. Although the average performance differences across various batch sizes were subtle, we noted that larger batch sizes led to a significantly higher variance in MSE, suggesting decreased consistency in model predictions.

Adding More Neurons

The complexity of a neural network, influenced by its number of neurons and layers, significantly affects its ability to model complex functions. An increase in the number of neurons

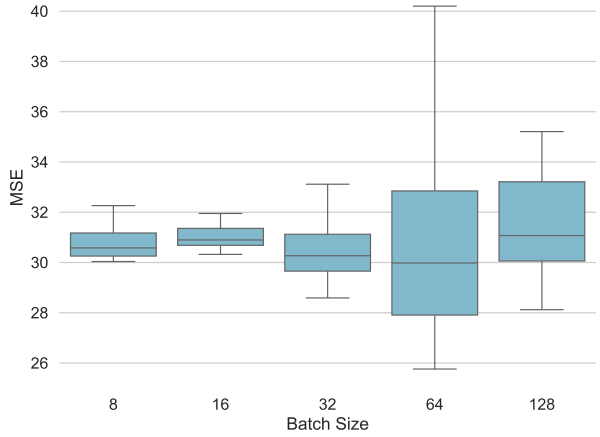


Figure 4

Model performance with different training batch sizes.

enhances the network's capacity to learn intricate patterns, aiding in more accurate function approximation. However, an excess of neurons can lead to overfitting, where the model excessively learns the idiosyncrasies of the training data, including noise and anomalies, resulting in poor performance on new, unseen data. Moreover, a larger number of neurons increases the model's computational complexity, extending training times and demanding more computational resources. The ideal number of neurons should be determined considering the task complexity, available data volume, and the desired equilibrium between model generalization and specificity (Qiao et al., 2017).

We start by investigating a single-hidden-layer FNN model with different numbers of neurons. Specifically, we increased the number of neurons from 4 to 8, 32, 64, and 128. As shown in Figure 5, as the neuron count increases, there is a continuous increase in MSE on the validation set. This suggests that for simpler datasets, a single-layer FNN with a lower neuron count may be adequate.

Adding More Layers

Our investigation extended to the influence of augmenting the number of hidden layers in the FNN. We tested configurations comprising 1, 2, and 3 hidden layers, each with 4, 32, and 128 neurons. In each model, all hidden layers contained an equal number of neurons. For instance, in the 4-neuron scenario, the configurations included a single-layer model with 4 neurons, a two-layer model with 4×4 neurons, and a three-layer model with $4 \times 4 \times 4$ neurons. This exploration led to a total of 9 distinct conditions.

The outcomes are illustrated in Figure 7. In scenarios with 4 neurons per layer, the MSE exhibits minimal variation across the different layer counts. This suggests that even a single-layer FNN is sufficiently complex for capturing the underlying patterns in the data. A similar trend is observed in the 32-neuron per-layer setups. With 128 neurons per layer, an increase in the number of layers seems to correlate with a decrease in MSE, indicating enhanced model performance. However, the configuration with 3 layers and 128 neurons per layer shows comparable performance

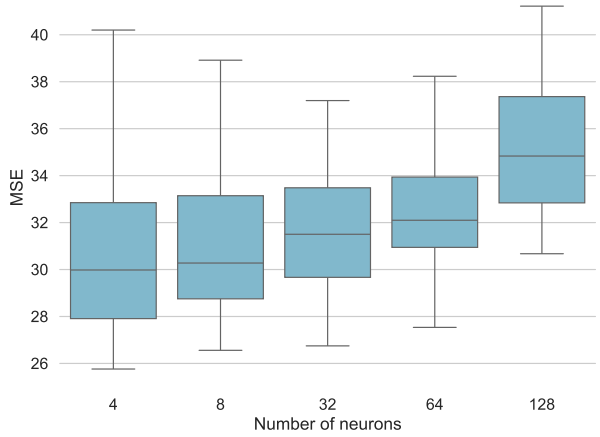


Figure 5

Model performance with different neuron counts in the hidden layer.

with models having 2 layers and 4 neurons per layer. These observations suggest that increasing network depth is not necessarily associated with improved predictive accuracy.

Grid Search

Having explored various hyperparameters and their impact on model performance, the question arises: how can we identify an effective combination of these parameters? One common technique is *grid search*, which employs an exhaustive search strategy to explore a predefined space of hyperparameter combinations by constructing a grid (Hutter et al., 2019). Each point on the grid represents a unique set of hyperparameters. By evaluating the model's performance for each combination, grid search identifies the optimal set that yields the best performance according to a predefined metric, such as MSE. This approach ensures that all possible combinations within the specified range are considered, providing a comprehensive picture of how different hyperparameters affect a model's performance. Grid search is particularly suitable for the psychological research in which the sample size is often not too big.

When optimizing the FNN using our example data, we implemented grid search by iterating over various learning rates (.1, .01, .001, .0001, and .00001), batch sizes (8, 16, 32, 64, and 128), and numbers of neurons (4, 8, 32, 64, and 128) and layers (1, 2, and 3). Table 2 shows the top 10 models sorted by the lowest mean MSE. The best model configuration simply based on MSE consists of 2 layers with 64 neurons each, a learning rate of 0.0001, and a batch size of 8. This model converged in 89 out of 100 replications. Among these successful cases, the mean MSE was 30.597, and the average number of training epochs required for convergence was 93. However, note that differences in MSE among different specifications in the table were not big, and the selection of the final model is eventually a preference of the researcher.

Besides prediction accuracy and training efficiency, researchers should also pay attention to the convergence issue when selecting models. For instance, the 10th-ranked model had a convergence rate of only 1%, i.e., it converged once in 100 replications. Despite its high accuracy in this

Table 2

Top 10 models from grid search (with 3 predictors). CR: convergence rate. MSE and the number of training epochs are based on converged cases.

ID	# layers	# neurons	Learning rate	Batch size	CR (%)	MSE		# training epochs	
						mean	std.	mean	std.
1	2	64	0.0001	8	89	30.597	0.983	93.011	21.506
2	3	32	0.001	64	71	30.666	5.741	49.648	14.344
3	1	4	0.001	32	83	30.700	1.771	44.940	15.607
4	3	64	0.0001	16	74	30.742	0.789	84.622	19.176
5	1	32	0.001	64	93	30.751	5.526	49.441	16.907
6	3	64	0.0001	8	70	30.762	1.611	71.614	21.716
7	2	128	0.00001	8	93	30.764	1.087	394.108	78.920
8	3	4	0.001	64	57	30.769	4.532	52.386	29.186
9	2	128	0.0001	8	93	30.782	2.362	84.538	20.004
10	3	64	0.1	128	1	30.790	-	26.000	-

one case, such a low convergence rate undermines the model's reliability and generalizability, and thus should not be selected. This example emphasizes the importance of evaluating models not only on their performance metrics, but also on their stability and consistency across multiple runs.

Convergence Rate

We further examine the effect of hyperparameters on model convergence. Figure 6 illustrates a case of non-convergence during training, where the training loss stagnated at an early stage, and the validation loss exhibited substantial fluctuations, suggesting being trapped in a local minimum. The convergence rates for different hyperparameter configurations are shown in Fig 8. Specifically, Fig 8a shows a clear trend where the convergence rate initially increases with the learning rate, but decreases as the learning rate continues to increase. This pattern is similar to the association between learning rate and MSE. On the other hand, as shown in Fig 8b, the effect of increasing the batch size seems minimal and generally results in a slight decrease in the probability of convergence. Meanwhile, Figure 8c shows that adding more neurons per layer tends to help with model convergence in general; however, all three plots suggest that adding more layers tends to decrease the likelihood of model convergence.

Other Influencing Factors

In addition to the hyperparameters that are directly related to a FNN model, there are other factors that can influence the performance of a model.

Number of Predictors

Incorporating more relevant predictors will generally improve the model's performance. To illustrate this effect, we extended our initial three predictors (age, education, and sex) to ten predictors: age, education, sex, site, booster, ufov, mmse, hvltt, hvltt2, and hvltt3. The performance

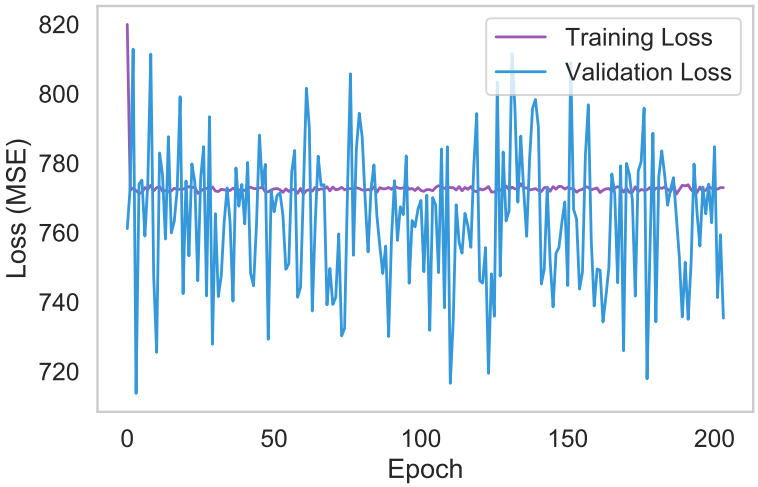


Figure 6

Example of non-convergence.

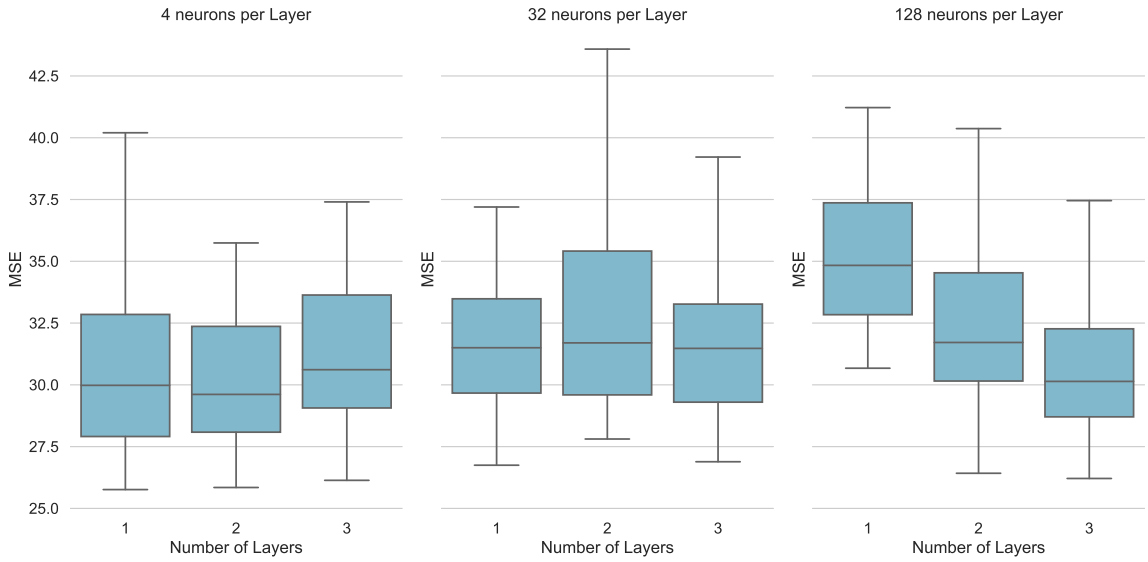
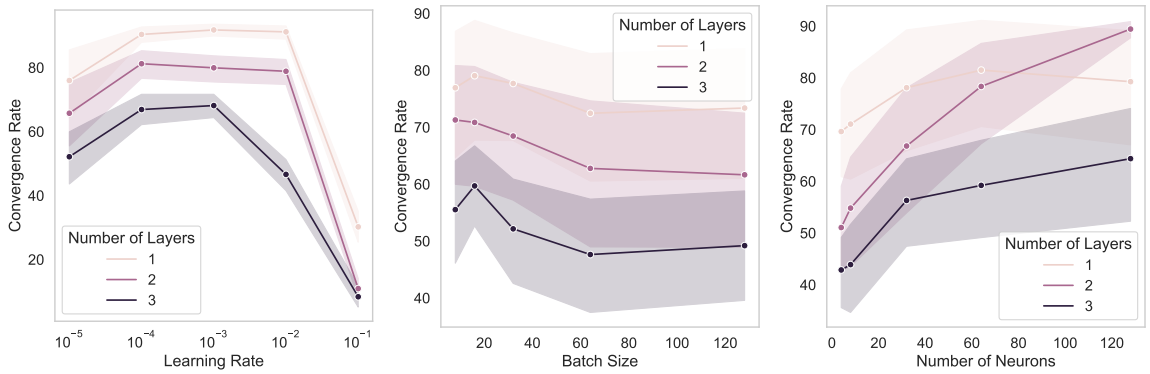


Figure 7

Model performance with different numbers of layers.



(a) *Learning rate.* (b) *Batch size.* (c) *Number of neurons.*

Figure 8

Model convergence rate across different hyperparameter settings.

outcomes are shown in Table 3. The optimal model converged in 90% cases with a mean MSE of 17.430. When compared to the results presented in Table 2, it is evident that adding these predictors substantially reduced the MSE and increased the convergence rate. Additionally, we can observe a trend towards simpler models being more effective with the extended predictor set: all of the ten best models consisted of only one hidden layer. This observation indicates that the selection of model hyperparameters can vary greatly depending on the particular circumstances.

Sample Sizes

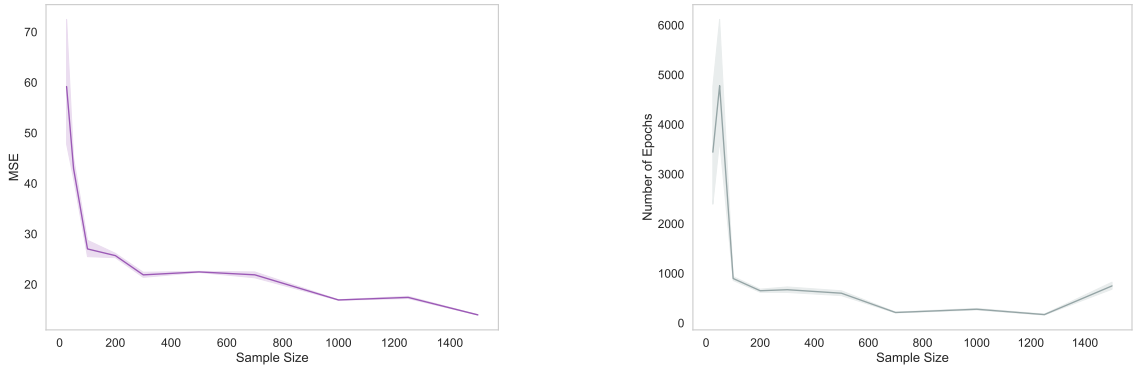
Number of samples in the dataset can significantly affect the prediction results of a neural network. Larger sample sizes typically yield better performance by providing a more comprehensive

Table 3

Top 10 models from grid search (with 10 predictors). CR: convergence rate. MSE and the number of training epochs are based on converged cases.

ID	# layers	# neurons	Learning rate	Batch size	CR (%)	MSE		# training epochs	
						mean	std.	mean	std.
1	1	4	0.0001	8	90	17.430	1.023	92.378	28.080
2	1	4	0.001	16	88	17.447	0.710	46.466	22.773
3	1	4	0.00001	8	82	17.464	1.692	414.354	154.924
4	1	4	0.001	64	91	17.525	2.956	56.198	16.578
5	1	4	0.0001	16	90	17.527	0.650	108.289	30.446
6	1	8	0.0001	8	93	17.599	0.918	103.387	28.296
7	1	4	0.001	128	86	17.638	0.795	73.849	18.637
8	1	4	0.001	8	85	17.656	0.696	35.894	11.484
9	1	4	0.0001	32	79	17.661	1.344	110.190	25.181
10	1	4	0.001	32	85	17.689	1.652	52.894	15.217

representation of the data distribution, while smaller sizes may lead to outfitting due to insufficient data. To test this, we built seven datasets with varied sample sizes (N = 25, 50, 100, 200, 300, 500, 700, 1,000, 1250, and 1,500) by randomly drawing subsets of samples from the original dataset. As we did before, each dataset was then randomly divided into training/evaluation/test sets with a 72:13:15 ratio. For every distinct sample size, we trained 100 separate FNN models with the default configurations and evaluated their performance. Note that we did the 100 replications for each different sample size on a fixed subset of samples instead of 100 random subsets of samples. This was to avoid importing extra randomness from different datasets.



(a) *MSE.* (b) *Number of epochs.*

Figure 9

Model performance with different sample sizes.

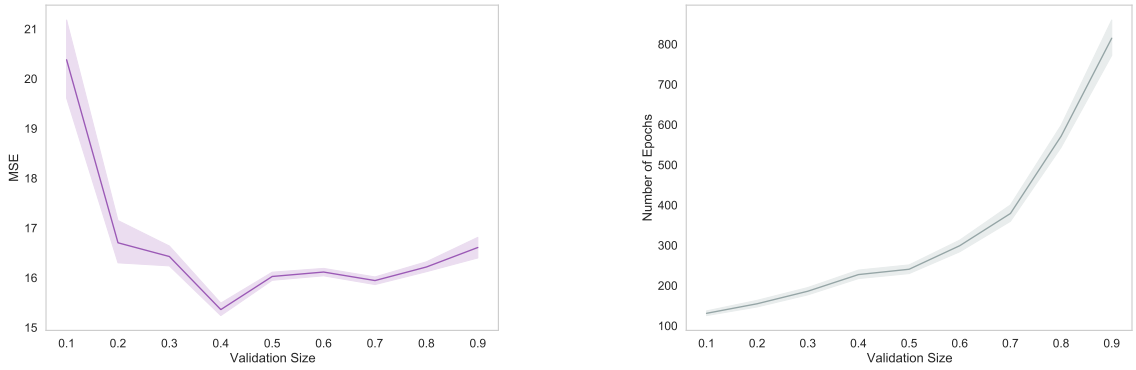
The model was trained using the 10 predictors outlined previously. Figure 9a illustrates the variation in MSE across different sample sizes. With the smallest sample size ($N = 25$), the MSE exhibited a high mean and high variance, indicating that the dataset was too limited for effective training. Increasing the sample size from 25 to 50 led to a notable decrease in MSE. Further increasing the sample size continued to lower the mean MSE and slightly reduced its variance, thereby enhancing the model's accuracy and consistency. Regarding the number of training epochs, as depicted in Figure 9b, very small datasets required an exceedingly large number of epochs since the model struggled to learn from the scant data, resulting in instability. As the sample size grew, the required number of training epochs diminished, reflecting improved learning efficiency. However, the number of epochs rose again with very large datasets, probably attributed to the complexity of learning from a large pool of data.

Training and Validation Set Sizes

The way in which the data is divided into a training set and a validation set can also affect the performance of the model. Specifically, allocating too large a portion of the dataset to the validation set can prevent the model from getting enough training data, which can lead to underfitting, i.e., the model is unable to effectively capture the underlying patterns in the data. Conversely, allocating too small a portion to the validation set may cause large variation in the validation step, thus preventing accurate optimization judgments such as when to terminate training and which model to choose. To empirically demonstrate these effects, we kept 15% of the original data as the test set in our

experiments and varied the ratio of the training set to the validation set so that the validation set accounted for 10%, 20%, 30%, . . . , 90% of the rest of the dataset, respectively.

Figure 10 shows the results. When the validation set comprises only 10% of the visible dataset (i.e., the combination of training and validation set), it contains $1,573 \times 0.85 \times 0.1 = 133$ participants, which is insufficient to accurately reflect the model's performance, resulting in inappropriate early stopping and a high MSE. As the proportion of the validation set increases to 0.4, the MSE decreases, but beyond that point, the MSE starts to increase again, which is mainly caused by insufficient training data. This suggests that for this particular dataset and the FNN model, allocating 40% of the visible data to the validation set is likely the optimal choice. Notably, Figure 10b shows a positive correlation between validation set size and the number of epochs required for training. One plausible explanation is that a smaller training set contains less information from the predictors, which might cause underfitting and require more epochs to converge.



(a) *MSE.* (b) *Number of epochs.*

Figure 10

Model performance with different validation set sizes.

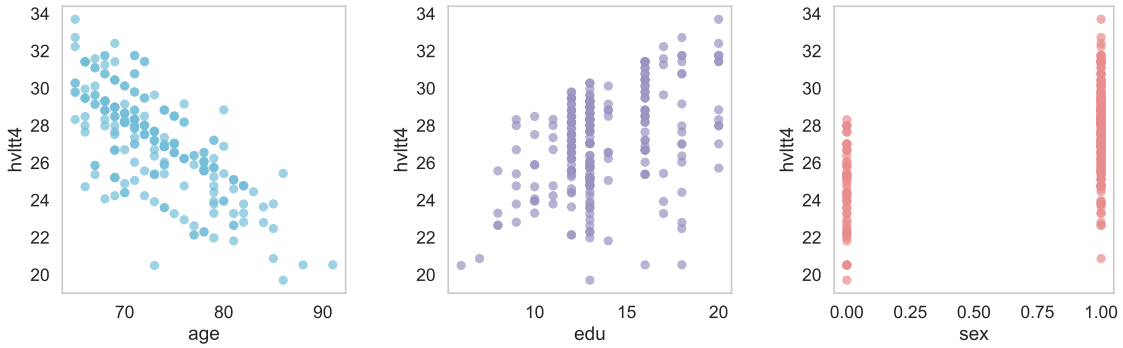
Model Interpretation

Interpreting the outcomes of deep learning models is crucial yet difficult, particularly in fields like psychology and other social sciences, where understanding the model's predictions can offer valuable insights into human behaviors and decision-making processes (Hassija et al., 2024; Orrù et al., 2020; Ribeiro et al., 2016; Rudin, 2019). This section outlines two simple, accessible methods for interpreting model results.

One common approach for model interpretation is partial dependence plots (PDPs), which visualize the relationship between the target variable and a subset of input features of interest while marginalizing over the values of all other input features (Hastie et al., 2009; Pedregosa et al., 2011). Intuitively, partial dependence can be interpreted as the expected target response as a function of the selected input features. By plotting the partial dependence, one can gain insights into the marginal effect of the input features on the model's predictions.

Figure 11 shows three one-way PDPs, each plotting a single input variable against the predicted outcome. From Figure 11a, we observe a trend where the model predicts better test performance with younger participants. Similarly, Figure 11b indicates that in general, participants

with longer education years are predicted to perform better in tests. Additionally, Figure 11c reveals a tendency of the model to predict higher test scores for female participants. These plots facilitate a potential understanding of the relationships between different input variables and outcomes.



(a) *Age.* (b) *Education.* (c) *Sex.*

Figure 11

One-way partial dependence plots (PDPs).

In addition to visualization, there are various methods to assess feature importance in neural network models (Molnar, 2020). A straightforward technique is feature permutation, i.e., randomly shuffling the values of each input variable and observing the impact on the network's performance (Breiman, 2001). A notable increase in MSE upon permuting a variable suggests its significant influence on the model's predictions. This process is repeated for each input variable (or subset of variables) to determine their relative importance.

In our study using the ACTIVE dataset with 10 variables, we observed distinct changes in model performance upon permutation. We conducted 500 replications, where in each replication, we randomly split the data into training, validation, and test sets. During the testing stage, we first tested the model on the original test set, then permuted each predictor in the test set and allowed the model to make predictions. As shown in Figure 12, permuting *hvltt3*, *hvltt*, and *hvltt2* led to the most substantial increase compared to the original average MSE (the red dotted line), indicating that previous test scores could be the most substantial predictors for accurately predicting the latest test score. Permuting the demographic variables *sex* and *age* also led to a slight increase in MSE, indicating their importance. The remaining predictors did not result in significant differences, which may suggest either that they are correlated with other predictors and thus contribute little additional information, or that they are irrelevant.

Discussion

In this paper, we presented a step-by-step guide for employing FNNs in psychological research. We started with a concrete example of a basic 3-layer FNN on a psychological dataset, including the matrix representation, the forward propagation, and the backpropagation process. Subsequent experiments demonstrated the impact of hyperparameter adjustments and network structure variations on model performance. Additionally, we showed two simple ways of interpreting neural network predictions and assessing feature importance.

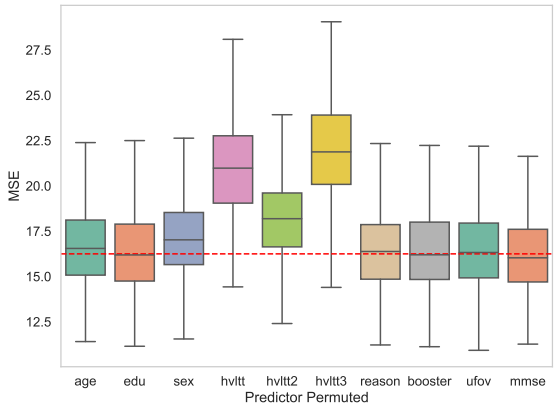


Figure 12

MSE with permuted predictors.

FNNs are the simplest form of neural networks. Subsequent innovation builds on its foundation but introduces new architecture and mechanisms to handle more complex data patterns, especially in sequence and spatial recognition tasks. These advancements allow for better performance in tasks involving sequences (like text and audio) and spatial data (like images). Specifically, Recurrent Neural Networks (RNNs, Rumelhart et al., 1985) and Convolutional Neural Networks (CNNs, LeCun et al., 1989) evolved from FNNs. RNNs can handle sequences by maintaining a memory of previous inputs, while CNNs excel in pattern recognition within image data through convolutional layers. Transformers (Vaswani et al., 2017), including models like BERT (Devlin et al., 2018) and the GPT series (Brown et al., 2020; Radford et al., 2019), represent a further advancement, introducing the self-attention mechanism. This allows them to process long-range dependencies and context more effectively, especially in complex tasks like language understanding and generation. However, it is important to note that these advanced models still rest on the foundational principles of FNNs. The understanding of core concepts, such as backpropagation, loss functions, and the intricacies of training and testing, is universal across these neural network architectures. The hyperparameter tuning strategies for FNNs are also applicable to more advanced models.

Although not considered in this tutorial, the type of loss functions, depending on the task type, also play an important role. Our primary task in this paper was regression, and we employed MSE as the loss function. However, when dealing with categorical data, other loss functions are more appropriate. For example, in classification tasks, cross-entropy loss is commonly used (Goodfellow et al., 2016; Hastie et al., 2009). In binary classification scenarios, binary cross-entropy is effective (Ruby & Yendapalli, 2020), while for multi-class problems, categorical cross-entropy and its variations are more suitable (Gordon-Rodriguez et al., 2020; Rusiecki, 2019). These loss functions are better aligned with the nature of categorical data and help accurately model the probability distribution of the class labels.

Finally, our experiments employed *supervised learning* (Hastie et al., 2009), as we had labeled training data, i.e., the actual values of the outcome variables. However, there are research scenarios where *unsupervised learning* could be beneficial (Xu & Wunsch, 2005). For instance, tasks like topic modeling (Blei et al., 2003) or clustering (Jain, 2010), which aim to discover inherent patterns or groupings in the data without pre-defined labels, can be approached with unsupervised

learning techniques (Ghahramani, 2003; Suominen & Toivanen, 2016).

Acknowldegment

This research was upported by the Institute of Education Sciences (R305D210023) the Lucy Family Institute for Data and Society and Notre Dame Global.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., et al. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*.

Agarap, A. F. (2018). Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*.

Bengio, Y. (2012). Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade: Second edition* (pp. 437–478). Springer.

Blei, D. M., Ng, A. Y., & Jordan, M. I. (2003). Latent dirichlet allocation. *Journal of machine Learning research*, *3*(Jan), 993–1022.

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

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, *33*, 1877–1901.

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Gareth, J., Daniela, W., Trevor, H., & Robert, T. (2013). *An introduction to statistical learning: With applications in r*. Spinger.

Ghahramani, Z. (2003). Unsupervised learning. In *Summer school on machine learning* (pp. 72–112). Springer.

Goldberg, Y. (2022). *Neural network methods for natural language processing*. Springer Nature.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT press.

Gordon-Rodriguez, E., Loaiza-Ganem, G., Pleiss, G., & Cunningham, J. P. (2020). Uses and abuses of the cross-entropy loss: Case studies in modern deep learning.

Hassija, V., Chamola, V., Mahapatra, A., Singal, A., Goel, D., Huang, K., Scardapane, S., Spinelli, I., Mahmud, M., & Hussain, A. (2024). Interpreting black-box models: A review on explainable artificial intelligence. *Cognitive Computation*, *16*(1), 45–74.

Hastie, T., Tibshirani, R., Friedman, J. H., & Friedman, J. H. (2009). *The elements of statistical learning: Data mining, inference, and prediction* (Vol. 2). Springer.

Hutter, F., Kotthoff, L., & Vanschoren, J. (2019). *Automated machine learning: Methods, systems, challenges*. Springer Nature.

Hyndman, R. J., & Koehler, A. B. (2006). Another look at measures of forecast accuracy. *International journal of forecasting*, *22*(4), 679–688.

Iyortsuun, N. K., Kim, S.-H., Jhon, M., Yang, H.-J., & Pant, S. (2023). A review of machine learning and deep learning approaches on mental health diagnosis. *Healthcare*, *11*(3), 285.

Jain, A. K. (2010). Data clustering: 50 years beyond k-means. *Pattern recognition letters*, *31*(8), 651–666.

Jobe, J. B., Smith, D. M., Ball, K., Tennstedt, S. L., Marsiske, M., Willis, S. L., Rebok, G. W., Morris, J. N., Helmers, K. F., Leveck, M. D., et al. (2001). Active: A cognitive intervention trial to promote independence in older adults. *Controlled clinical trials*, *22*(4), 453–479.

Kaddour, J., Harris, J., Mozes, M., Bradley, H., Raileanu, R., & McHardy, R. (2023). Challenges and applications of large language models. *arXiv preprint arXiv:2307.10169*.

Khan, S., Rahmani, H., Shah, S. A. A., Bennamoun, M., Medioni, G., & Dickinson, S. (2018). *A guide to convolutional neural networks for computer vision* (Vol. 8). Springer.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *nature*, *521*(7553), 436–444.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural computation*, *1*(4), 541–551.

LeCun, Y., Bottou, L., Orr, G. B., & Müller, K.-R. (2002). Efficient backprop. In *Neural networks: Tricks of the trade* (pp. 9–50). Springer.

Malhotra, A., & Jindal, R. (2022). Deep learning techniques for suicide and depression detection from online social media: A scoping review. *Applied Soft Computing*, 109713.

Masters, D., & Luschi, C. (2018). Revisiting small batch training for deep neural networks. *arXiv preprint arXiv:1804.07612*.

Miles, J. (2005). R-squared, adjusted r-squared. *Encyclopedia of statistics in behavioral science*.

Min, S., Lee, B., & Yoon, S. (2017). Deep learning in bioinformatics. *Briefings in bioinformatics*, *18*(5), 851–869.

Molnar, C. (2020). *Interpretable machine learning* (3rd). Christoph Molnar. https://christophm.github.io/interpretable-ml-book/

Orrù, G., Monaro, M., Conversano, C., Gemignani, A., & Sartori, G. (2020). Machine learning in psychometrics and psychological research. *Frontiers in psychology*, *10*, 492685.

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

Powers, D. M. (2020). Evaluation: From precision, recall and f-measure to roc, informedness, markedness and correlation. *arXiv preprint arXiv:2010.16061*.

Qiao, J., Li, S., Han, H., & Wang, D. (2017). An improved algorithm for building self-organizing feedforward neural networks. *Neurocomputing*, *262*, 28–40.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. (2019). Language models are unsupervised multitask learners. *OpenAI blog*, *1*(8), 9.

Radiuk, P. M. (2017). Impact of training set batch size on the performance of convolutional neural networks for diverse datasets. *Information Technology and Management Science*, *20*(1), 20–24.

Ribeiro, M. T., Singh, S., & Guestrin, C. (2016). "why should i trust you?" explaining the predictions of any classifier. *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 1135–1144.

Richards, B. A., Lillicrap, T. P., Beaudoin, P., Bengio, Y., Bogacz, R., Christensen, A., Clopath, C., Costa, R. P., de Berker, A., Ganguli, S., et al. (2019). A deep learning framework for neuroscience. *Nature neuroscience*, *22*(11), 1761–1770.

Ruby, U., & Yendapalli, V. (2020). Binary cross entropy with deep learning technique for image classification. *International Journal of Advanced Trends in Computer Science and Engineering*, *9*(4).

Rudin, C. (2019). Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature machine intelligence*, *1*(5), 206–215.

Rumelhart, D. E., Hinton, G. E., Williams, R. J., et al. (1985). Learning internal representations by error propagation.

Rusiecki, A. (2019). Trimmed categorical cross-entropy for deep learning with label noise. *Electronics Letters*, *55*(6), 319–320.

Sazli, M. H. (2006). A brief review of feed-forward neural networks. *Communications Faculty of Sciences University of Ankara Series A2-A3 Physical Sciences and Engineering*, *50*(01), 11–17.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural networks*, *61*, 85–117.

Su, C., Xu, Z., Pathak, J., & Wang, F. (2020). Deep learning in mental health outcome research: A scoping review. *Translational Psychiatry*, *10*(1), 116.

Suominen, A., & Toivanen, H. (2016). Map of science with topic modeling: Comparison of unsupervised learning and human-assigned subject classification. *Journal of the Association for Information Science and Technology*, *67*(10), 2464–2476.

Tadesse, M. M., Lin, H., Xu, B., & Yang, L. (2019). Detection of suicide ideation in social media forums using deep learning. *Algorithms*, *13*(1), 7.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, *30*.

Wilson, D. R., & Martinez, T. R. (2003). The general inefficiency of batch training for gradient descent learning. *Neural networks*, *16*(10), 1429–1451.

Xu, R., & Wunsch, D. (2005). Survey of clustering algorithms. *IEEE Transactions on neural networks*, *16*(3), 645–678.

Yu, T., & Zhu, H. (2020). Hyper-parameter optimization: A review of algorithms and applications. *arXiv preprint arXiv:2003.05689*.

Zheng, A., & Casari, A. (2018). *Feature engineering for machine learning: Principles and techniques for data scientists*. O'Reilly Media, Inc.