MULTIVARIATE BEHAVIORAL RESEARCH
2026, VOL. 61, NO. 2, 335–355

OPEN ACCESS

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

Lingbo Tong[a] and Zhiyong Zhang[b]

[a]Department of Educational Psychology, University of Wisconsin–Madison, Madison, Wisconsin, USA; [b]Department of Psychology, University of Notre Dame, Notre Dame, Indiana, USA

ABSTRACT
Artificial neural networks (ANN) have attracted increasing 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 article presents a step-by-step guide for implementing a feed-forward neural network (FNN) on a psychological data set 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); machine learning; psychological data sets; stochastic gradient descent (SGD); hyperparameter optimization

Introduction

An *artificial neural network* (ANN), also referred to simply as a *neural network*, is a computational model inspired by the structure and functioning of the human brain (Bishop & Nasrabadi, 2006). It consists of interconnected units (or "neurons") organized into layers that transform and pass information forward through the network. Compared with the traditional statistical models such as linear regression, neural networks are particularly useful for modeling complex, nonlinear relationships in data.

Among the simplest forms of neural networks is the *feed-forward neural network* (FNN), where information flows in one direction, from the input layer through one or more hidden layers to the output layer, without loops or cycles. Even with a single hidden layer, FNNs are capable of approximating a wide range of functions, making them a foundational building block in modern machine learning (Schmidhuber, 2015).

When a neural network contains more than one hidden layer, it is commonly referred to as a *deep learning* (DL) model. The "deep" in deep learning refers to the depth of the architecture (multiple layers of transformations between input and output) that allows the model to learn complex representations of the data (LeCun et al., 2015). Deep learning builds on the same foundational principles as simpler neural networks, but extends them in depth and capacity, making them powerful in solving complex tasks in data-rich environments.

Deep learning has become 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, for example, 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 (Abdurahman et al., 2024).

The exponential growth of the internet has facilitated the collection of massive data sets, combined with the advent of computing resources, granting scientists opportunities to create and distribute large-scale

© 2025 The Author(s).
This is an Open Access article distributed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits non-commercial re-use, distribution, and reproduction in any medium, provided the original work is properly cited, and is not altered, transformed, or built upon in any way. The terms on which this article has been published allow the posting of the Accepted Manuscript in a repository by the author(s) or with their consent.

deep learning models (Abadi et al., 2016; Kaddour et al., 2023). However, the current research landscape seems oriented toward these large-scale models and extensive data sets, leaving a gap in exploring the efficacy of portable models in smaller, lab-collected data sets, 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 due to lack access to large data sets, computational resources, or training in machine learning methods (Yarkoni & Westfall, 2017).

While there are primers designed to introduce psychologists to deep learning (Urban and Gates (2021; Orrù et al. 2020), many of these resources assume familiarity with machine learning concepts or focus on high-level overviews rather than step-by-step model building and training. The gap becomes particularly evident when considering applied researchers with a background in regression modeling but limited experience with neural network implementations. 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 article 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 FNN to the data set 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.

Data set

This article's data are drawn from the ACTIVE data set (Jobe et al., 2001). Originating from a large-scale study conducted from 1999 to 2001 in the United States, the ACTIVE data set 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 data set comprises 1,573 participants.[1]

In the examples throughout this article, we use `hvltt4` as the model outcome. Our primary research question is how accurately this outcome can be predicted using the remaining variables in the data set. We use this predictive task to demonstrate neural network training techniques and hyperparameter tuning strategies.

Feed-Forward neural network

A feed-forward neural network (FNN), also known as a multilayer perceptron (MLP), 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 computational units (often called "neurons" or "nodes") that receive inputs, apply transformations using learned parameters (weights and biases), and pass outputs forward to subsequent layers. In this paper, we use the term "neurons" to refer specifically to these computational units or nodes, rather than to individual parameters.

The simplest FNN contains three layers: an input layer, a hidden layer, and an output layer. Figure 1 illustrates such structure for a single observation *i*. In practice, we typically feed multiple observations simultaneously into the FNN *via* matrix multiplication.

Mathematically, the model can be defined as

$$\hat{\mathbf{Y}} = \underbrace{\left[\phi\left(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{1}\mathbf{b}^{(1)}\right)\right]}_{\text{Hidden layer output}} \mathbf{W}^{(2)} + \mathbf{1}\mathbf{b}^{(2)}. \tag{1}$$

Here, $\mathbf{X}$ is the input data matrix, $\mathbf{W}^{(1)}$ denotes the weight matrix connecting the input layer and the hidden layer, and $\mathbf{b}^{(1)}$ is the corresponding row vector of bias parameters for the hidden layer. $\phi(\cdot)$ represents an activation function. Similarly, $\mathbf{W}^{(2)}$ denotes the

[1]The dataset and the code are available at https://osf.io/pwr8t.

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

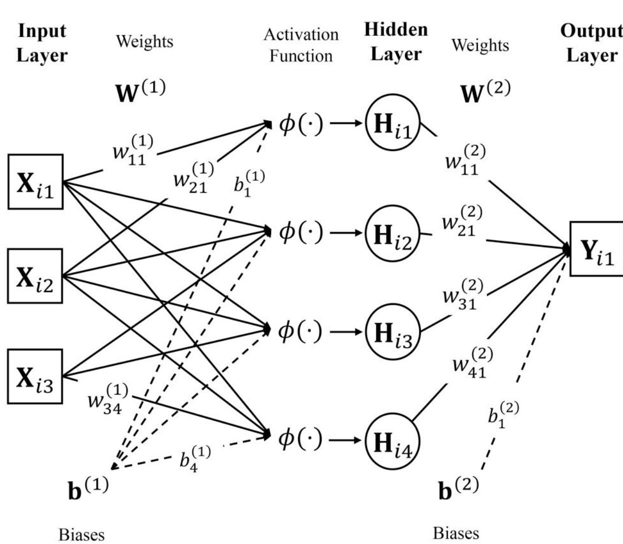


Figure 1. Illustration of a three-layer FNN for a single observation *i*. Since the task involves predicting a continuous outcome, the output layer uses an identity activation function (not explicitly shown here).

weight matrix connecting the hidden layer to the output layer, and $\mathbf{b}^{(2)}$ is the associated row vector of biases for the output layer. $\hat{\mathbf{Y}}$ 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 in subsequent subsections, using the ACTIVE data example. Suppose in the example, we want to predict the verbal test score at time 4 (hvltt4) using age, years of education, and gender.

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 raw input data matrix, denoted as $\mathbf{X}_{raw}$, appears as follows with the collected data in the ACTIVE study:

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

Here, x_{ij} indicates the value for the ith participant of the jth predictor. For instance, the first row in the input matrix represents a record from a 65-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, that is, 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 variance (Zheng & Casari, 2018). Furthermore, it helps gradient-based optimization algorithms converge faster and more effectively during training (Goodfellow et al., 2016). After normalization, the standardized input data matrix, denoted as $\mathbf{X}$, becomes:

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

Hidden layer

Hidden layers in FNNs can be conceptually compared to latent variables in structural equation modeling. While latent variables represent the theoretical constructs specified by the researcher, hidden layers learn the data-driven representations optimized for predictive accuracy.

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 $\mathbf{H} = \phi(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{1}\mathbf{b}^{(1)})$, where $\mathbf{X}$ is an $n \times p$ matrix representing the input data from the previous layer, $\mathbf{W}^{(1)}$ is a $p \times h$ weight matrix denoting the linear transformation applied to the input data, and $\mathbf{b}^{(1)}$ is defined as a $1 \times h$ row vector. Here, $\mathbf{1}$ is an $n \times 1$ column vector of ones, and the multiplication $\mathbf{1}\mathbf{b}^{(1)}$ ensures the bias vector is correctly added (or *broadcasted*) to each row of the matrix product $\mathbf{X}\mathbf{W}^{(1)}$. The activation function $\phi(\cdot)$ is then applied element-wise, yielding the transformed hidden layer output matrix $\mathbf{H}$, which has dimensions $n \times h$.

For instance, if $h = 4$, that is, the hidden layer comprises 4 neurons, the weights and biases would be structured as follows:

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

where $w_{ij}^{(1)}$ denotes the ith weight value for the jth neuron of the hidden layer, and $b_j^{(1)}$ denotes the bias associated with the jth neuron of that layer. During the linear summation, the input data matrix $\mathbf{X}$ is multiplied by the weight matrix $\mathbf{W}^{(1)}$, yielding a matrix of dimensions $n \times h$. Subsequently, the bias vector $\mathbf{b}^{(1)}$ is added to each row of the product matrix, a process called *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 Rectified Linear Unit (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. It is worth noting, however, that the Sigmoid function is commonly preferred in binary classification tasks, where its output range of 0 to 1 naturally represents a probability. For readers seeking a more in-depth comparison of activation functions and their roles in deep learning, we recommend Urban and Gates (2021).

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. The choice of activation function for the output layer depends on the nature of the prediction task. For problems where the target variable is continuous, 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 $\mathbf{Y} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{1}\mathbf{b}^{(2)}$. 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, that is, 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 ith participant concerning the jth dependent variable. 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

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

For our specific case, this calculation becomes $(3+1) \times 4 + (4+1) \times 1 = 21$ parameters. The following section explains how the model iteratively updates its parameters to estimate their optimal values.

Model training and evaluation

In traditional regression models, estimating model parameters usually means finding regression coefficients that minimize the sum of squared errors. In machine learning, this process is referred to as *model training*, which is conceptually analogous to parameter estimation. During training, the model learns from a data set by iteratively adjusting its parameters to minimize prediction errors. *Model evaluation*, in turn, assesses how well a model's predictions align with the actual outcomes.

Data split

Before beginning parameter estimation (training), we set aside a portion of the data set for evaluation purposes. This approach differs somewhat from traditional multivariate modeling techniques (e.g., structural equation modeling, SEM), where researchers typically utilize the entire data set to estimate parameters and then evaluate model fit using fit indices such as BIC, RMSEA, or CFI. In contrast, machine learning emphasizes the model's predictive performance on unseen data. Specifically, the primary goal of machine learning modeling is generalizability: the ability of a model to accurately predict outcomes on new data beyond the sample used for parameter estimation. Using separate subsets of the data to train, validate, and test the model ensures that evaluation metrics truly reflect predictive accuracy rather than merely describing model fit to the training sample.

A common practice in machine learning is to divide the original data set 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 the 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 data set 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 data set, respectively.

Training

Training an FNN is the process of finding the weights and biases in a neural network. It involves several key steps, including parameter initialization, forward propagation, calculation of the loss function, and backpropagation.

Parameter initialization

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 that receives a inputs from the previous layer, its weights and biases can be initialized from a uniform distribution $U\left(-\sqrt{k}, \sqrt{k}\right)$, where $k = \frac{1}{a}$, through random number generation. Note that a refers to the number of inputs to a given layer, which may differ from p, the number of input features initially provided by the researcher at the input layer.

In the context of our example, the initialized parameter matrices appear as follows:

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

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

Forward propagation

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

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

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:

$$\hat{\mathbf{Y}} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{1}\mathbf{b}^{(2)} = \begin{bmatrix} -3.2497 \\ -0.0871 \\ -1.2196 \\ -0.0290 \\ -2.5735 \end{bmatrix}. \tag{14}$$

Calculating the loss function

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 jth dependent variable for the ith 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 Equation (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 data set. 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.

Gradient descent and backpropagation

In traditional statistical modeling, such as maximum-likelihood estimation (MLE), parameters are estimated by maximizing the likelihood of the observed data under a specified model. To maximize the likelihood function, iterative algorithms are often used. Similarly, neural networks are typically trained using iterative optimization algorithms that minimize a loss function directly, without relying on an explicit probabilistic model.

One of the most widely used optimization techniques in this context is *gradient descent*. Gradient descent is an iterative optimization method that updates model parameters in the direction opposite to the gradient of the loss function. The gradient indicates how the loss changes with respect to each parameter; thus, moving in the negative gradient direction gradually reduces the loss. In its basic form, gradient descent goes over the entire training data set at each iteration, which can be computationally expensive, especially for large data sets.

Mathematically, the update rule for gradient descent is given by:

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

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 $\frac{\partial l}{\partial \theta_t}$ is the gradient of the model loss l with respect to $\boldsymbol{\theta}$.

In practice, we typically use a stochastic variant of this algorithm, known as *Stochastic Gradient Descent (SGD)*. Instead of going over the entire data set at each iteration, SGD takes a small, randomly selected batch of training examples. This approach is computationally more efficient and introduces helpful noise into the optimization process.

To compute these gradients efficiently in a neural network, we use the *backpropagation* algorithm. Backpropagation is a recursive method for efficiently computing the gradient of the loss function with respect to each parameter in the network. It works by propagating the error signal from the output layer backward through the network, layer by layer, computing the contribution of each parameter to the total error.

In short, gradient descent defines the optimization strategy, SGD offers a practical and scalable implementation of that strategy, and backpropagation provides an efficient way to compute the required gradients.

We now illustrate how to derive these gradients step by step using our example data set, starting with the mean squared error (MSE) 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, $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x}$. To illustrate clearly how the chain rule operates in matrix form, we first consider a general case explicitly. Suppose we have a scalar function g that depends on the matrix $\mathbf{U}$, where the matrix $\mathbf{U}$ itself is defined as a linear transformation of another matrix $\mathbf{V}$. Specifically, let $\mathbf{U} = \mathbf{VW} + \mathbf{C}$, with $\mathbf{C}$ representing a bias matrix. Under these conditions, the chain rule in matrix calculus notation is expressed as follows:

$$g = f(\mathbf{U}), \quad \mathbf{U} = \mathbf{VW} + \mathbf{C} \quad \Rightarrow \quad \frac{\partial g}{\partial \mathbf{V}} = \frac{\partial g}{\partial \mathbf{U}}\mathbf{W}^T, \quad \frac{\partial g}{\partial \mathbf{W}} = \mathbf{V}^T\frac{\partial g}{\partial \mathbf{U}}. \tag{22}$$

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 $\mathbf{Y} = \mathbf{HW}^{(2)} + \mathbf{b}^{(2)}$, 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:

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

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:

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

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

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

we can apply the chain rule to get:

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

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

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

and

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

Plugging in numerical values, we have:

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

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

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.

Batches, epochs, and early stopping

The five 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 minibatch is a subset of the training data set used for a single step of gradient calculation and weight updating. Compared with computing gradients over the entire data set, SGD with minibatches is more computational efficient, especially when working with large data sets, as it reduces the amount of data that must be processed at each step. It also helps stabilize and accelerate learning by introducing small noise into the gradient estimates. Because each minibatch is a different random sample, the gradient it produces is just a quick but imperfect guess of the true gradient across all data. This variation between steps creates a "noisy" path toward the optimal solution, which helps the model escape local minima and explore the error surface more effectively. One helpful analogy is to imagine a ball rolling down a bumpy landscape; the random jolts can provide the necessary kick to escape a small divot and continue toward a deeper valley.

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.

If the total number of training samples is not evenly divisible by the batch size, the final batch of each epoch will contain fewer samples than the specified batch size. This smaller batch is still used to compute gradients and update the model. Most deep learning frameworks handle this automatically. Alternative strategies include padding (i.e., adding synthetic or duplicated samples to the final batch to match the expected batch size) or discarding the last few samples, but using the remainder batch is the standard practice. For more comprehensive discussion on batch and SGD algorithm, see Goodfellow et al. (2016, Chapters 8.1–8.3).

Once the network has processed all the data in the training data set, batch by batch, this completes one training iteration, typically called an *epoch*. In the traning stage, the model repeatedly cycles through the full data set across multiple epochs. Reusing the data across epochs allows the model to gradually refine its parameter estimates through many small adjustments, rather than attempting to learn everything in a single pass. 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

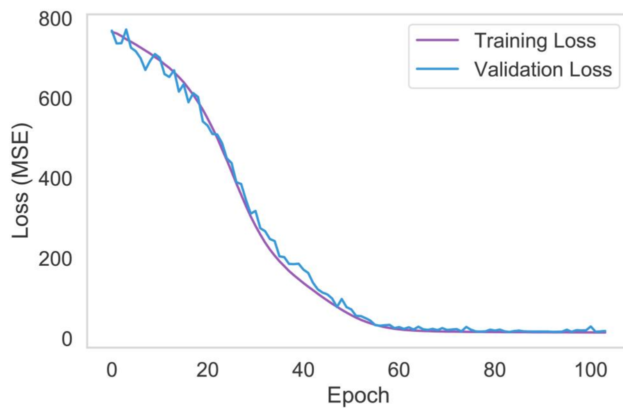


Figure 2. Change of MSE loss with the number of epochs during the training stage.

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.

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 data set, 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 data set, 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.

Software tools for neural network modeling

Several widely used software packages in both Python and R can facilitate the construction and training of neural networks. In Python, the most popular libraries include `PyTorch` (Paszke et al., 2019) and `TensorFlow` (Abadi et al., 2015), both of which offer flexible and powerful tools for defining, training, and evaluating neural networks. For those seeking higher-level APIs, `Keras` (Chollet et al., 2015), which runs on top of `TensorFlow`, provides a user-friendly interface for building models with minimal code. In this paper, we utilized `PyTorch` to construct the model and conduct all experiments.

In R, the `torch` (Falbel & Luraschi, 2025), `tensorflow` (Allaire & Tang, 2025), and `Keras` (Chollet et al., 2017) packages provide R interfaces to the corresponding Python libraries, allowing users to build deep learning models in R with similar functionality. Other R-based packages include `nnet` (Venables & Ripley, 2002) for simple feed-forward neural networks and `neuralnet` (Fritsch et al., 2020) for more customizable architectures, though these are more limited in scalability.

Researchers can choose the programming environment based on their familiarity and project needs. Python is typically favored for large-scale or production-level applications, while R is often chosen for integration with statistical workflows and data analysis pipelines.

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 data set.

Adjusting learning rate

The learning rate in SGD controls the step size of parameter updates toward 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 0.00001. 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 0.00001 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 described above.

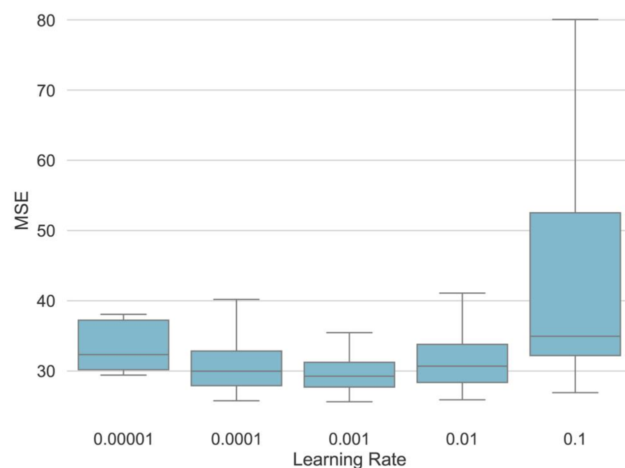


Figure 3. Model performance with different SGD learning rates.

Changing batch size

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

In contrast, 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 faster 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 requires 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 run time 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 data set's characteristics (Radiuk, 2017).

Batch sizes are commonly set to powers of two (e.g., 8, 16, 32, 64, 128) primarily due to the computational efficiencies. Modern GPUs and CPUs handle operations more efficiently when working with sizes aligned to powers of two, optimizing parallel processing

capabilities and memory management. When designing deep learning experiments, researchers typically set memory-related hyperparameters, such as batch sizes and the number of neurons per layer in the neural network, to powers of two for efficiency reasons and to simplify systematic exploration of the parameter space (Masters and Luschi (2018; Keskar et al. (2016).

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.

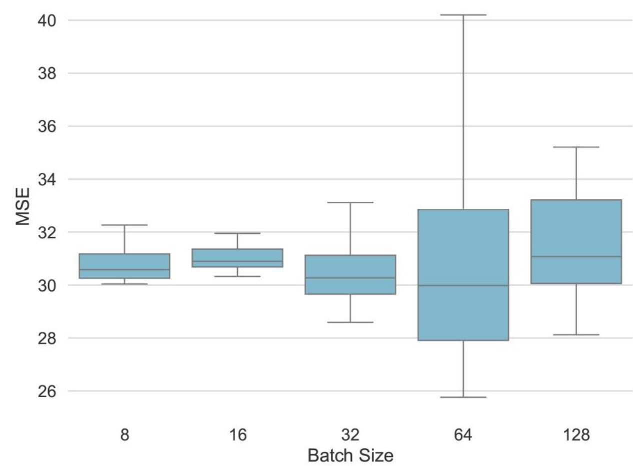


Figure 4. Model performance with different training batch sizes.

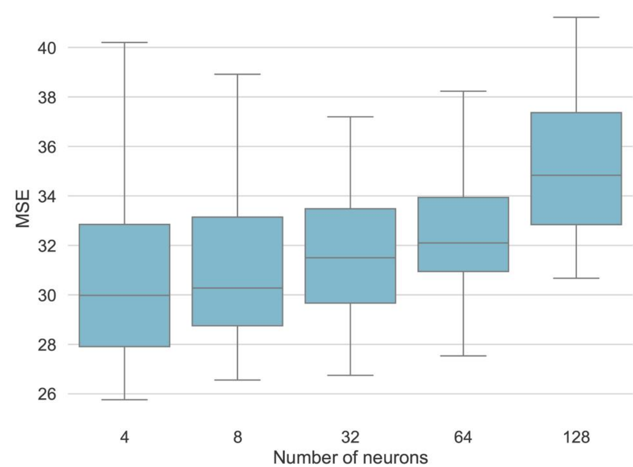


Figure 5. Model performance with different neuron counts in the hidden layer.

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 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 increase 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). Tasks involving more complex relationships or higher-dimensional input data may benefit from more hidden layers, while simpler tasks or smaller data sets may favor smaller architectures to mitigate overfitting. Increasing the number of neurons expands the model's capacity to approximate complex, nonlinear functions, which enhances specificity but can reduce generalizability if not matched to the complexity of the data. In practice, researchers often begin with a reasonable range of neuron counts and empirically evaluate performance using systematic methods, as we further explain in Section 5.5.

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 data sets, 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 four 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 nine distinct conditions.

The outcomes are illustrated in Figure 7. In scenarios with four 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 with models having 2 layers and 4 neurons per layer. These observations suggest that increasing network depth is not necessarily associated with the improved predictive accuracy.

Hyperparameter optimization

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.

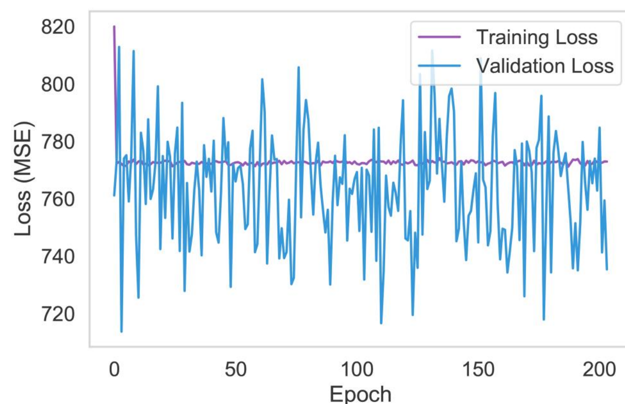


Figure 6. Example of non-convergence.

When optimizing the FNN using our example data, we implemented grid search by iterating over various learning rates (0.1, 0.01, 0.001, 0.0001, and 0.00001), 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, since the performance differences among the top models in the table are marginal, the final selection depends on researcher preference, which is often guided by principles like *model parsimony*. This criterion, commonly valued in the social sciences, favors the simplest model architecture when predictive

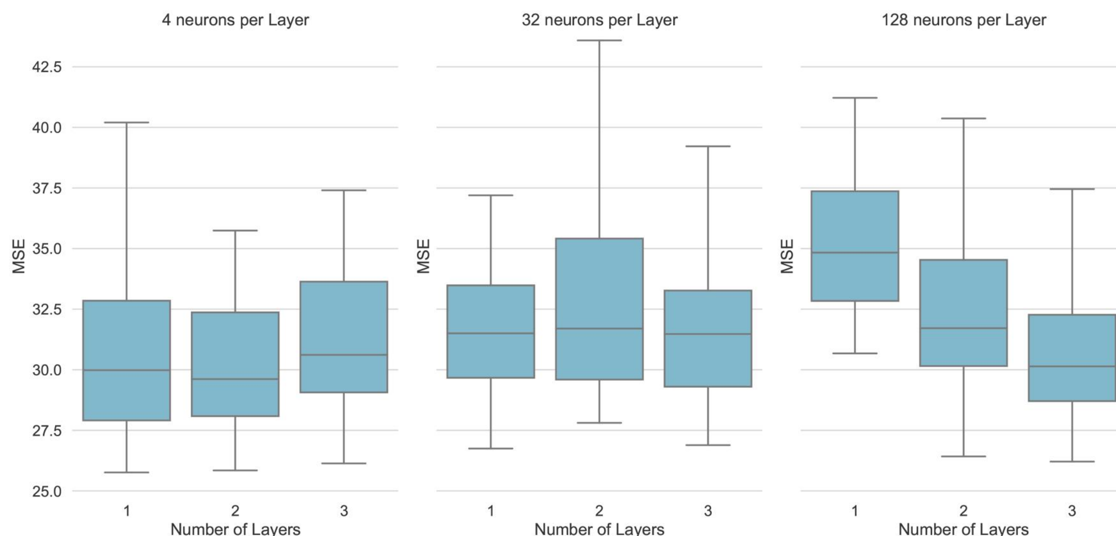
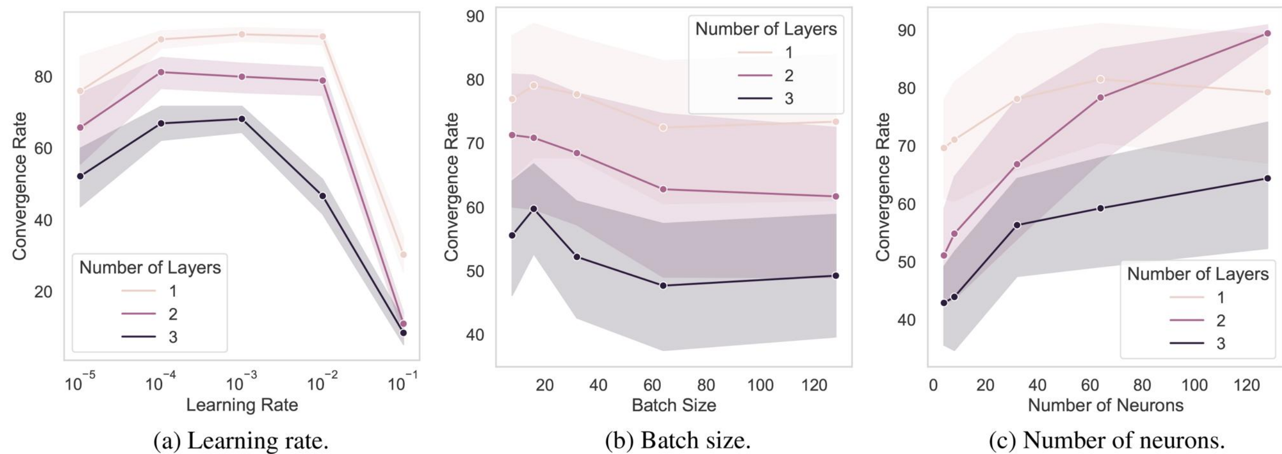


Figure 7. Model performance with different numbers of layers.



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

Figure 8. Model convergence rate across different hyperparameter settings.

accuracy is comparable across more complex alternatives.

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%; that is, it converged once in 100 replications. Despite its high accuracy in this 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.

Alternative optimization strategies are also widely used, especially when computational efficiency is a concern. For instantce, *Random search* (Bergstra & Bengio, 2012) samples combinations randomly within the specified ranges and has been shown to outperform grid search in high-dimensional spaces. More advanced techniques such as *Bayesian optimization* (Snoek et al., 2012) model the performance landscape and guide the search toward promising regions based on prior evaluations. These methods can be particularly useful when training models is expensive or the hyperparameter space is large.

In our case, grid search was implemented using base functions and custom scripts. For researchers seeking more scalable or automated solutions, several software packages are available. In R, `tune` (Kuhn, 2025) facilitates hyperparameter tuning for the tidymodels packages. `mlr3tuning` (Becker et al., 2025) finds optimal hyperparameter configurations for any 'mlr3' learner with several optimization algorithms, including grid search, random search, and Bayesian optimization. In Python, libraries such as `scikit-learn` (Pedregosa et al., 2011), `Optuna` (Akiba et al., 2019), `Hyperopt` (Bergstra et al., 2015), and `Ray Tune` (Liaw et al., 2018) support a variety of tuning strategies.

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 Figure 8. Specifically, Figure 8(a) 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 Figure 8(b), the effect of increasing the batch size seems minimal and generally results in a slight decrease in the probability of convergence. Meanwhile, Figure 8(c) 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

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	–

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

effect, we extended our initial 3 predictors (age, education, and sex) to 10 predictors: age, education, sex, site, booster, ufov, mmse, hvltt, hvltt2, and hvltt3. The performance 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.

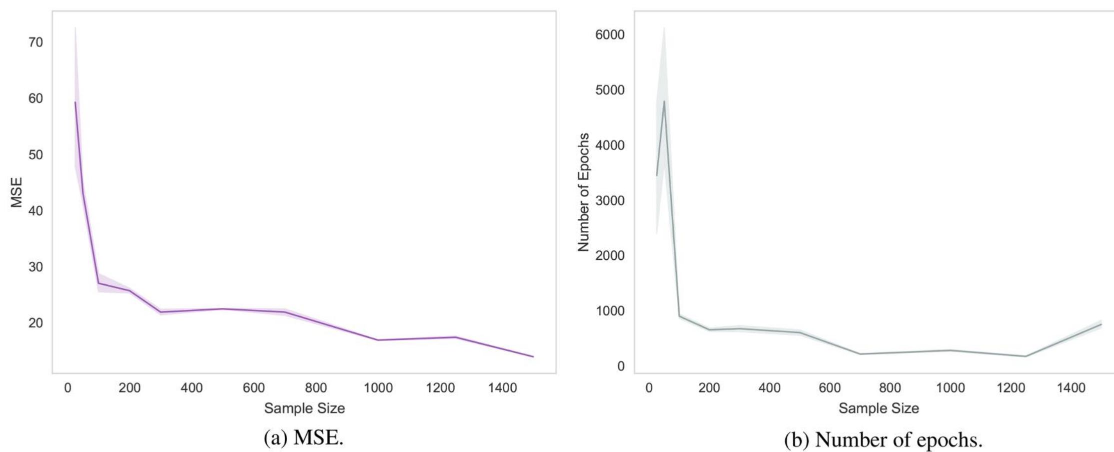
Interestingly, we also observed a shift toward simpler architectures: All of the 10 best-performing models in this extended setting consisted of only 1 hidden layer. This may indicate that the additional predictors made the underlying patterns easier to learn, reducing the need for deep intermediate representations. With richer input features, even shallow networks can capture the relevant relationships without relying on deeper architectures.

Moreover, deeper networks can sometimes introduce unnecessary complexity, which may hinder optimization and increase the risk of overfitting—especially when the feature set is limited. In fact, when comparing Table 2 with Table 3, it appears that the previous models trained with only three predictors may have been overfitting to the limited information available.

Sample sizes

Number of samples in the data set can significantly affect the prediction results of a neural network. Larger sample sizes typically yield better performance by providing a more comprehensive representation of the data distribution, while smaller sizes may lead to outfitting due to insufficient data. To test this, we built seven data sets 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 data set. As we did before, each data set 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 data sets.

The model was trained using the 10 predictors outlined previously. Figure 9(a) 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 data set was too limited for effective training. Increasing the sample



(a) MSE.

(b) Number of epochs.

Figure 9. Model performance with different sample sizes.

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 9(b), very small data sets 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 data sets, 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 data set to the validation set can prevent the model from getting enough training data, which can lead to underfitting; that is, 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. Here, we include a set of experiments that vary the ratio of training and validation set sizes to illustrate these effects. This is not part of the core modeling procedure, but is presented as additional context to help researchers remain mindful of data set composition when interpreting validation performance.

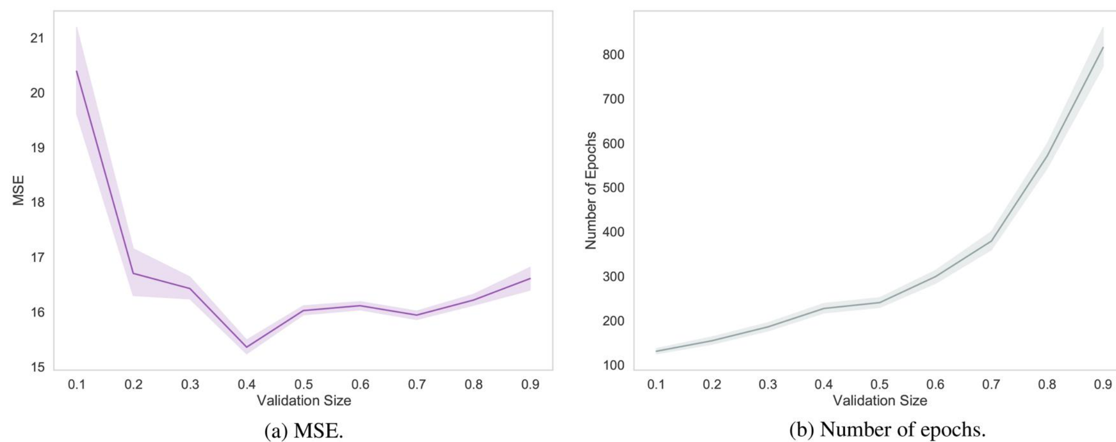
To empirically demonstrate these effects, we kept 15% of the original data as the test set in our experiments and varied the ratio of the training set to the validation set so that the validation set accounted for 10%, 20%, 30%, …, 90% of the rest of the data set, respectively.

Figure 10 shows the results. When the validation set comprises only 10% of the visible data set (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 data set and the FNN model, allocating 40% of the visible data to the validation set is likely the optimal choice. Notably, Figure 10(b) 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.

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 visualizes the relationship between the target variable and a subset



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

Figure 10. Model performance with different validation set sizes.

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. Unlike simple bivariate plots, which show raw relationships between two observed variables, PDPs depict the marginal effect of selected input features on the model's predictions while averaging out the influence of other variables in the model.

Figure 11 shows three one-way PDPs, each plotting a single input variable against the predicted outcome. From Figure 11(a), we observe a trend where the model predicts better test performance with younger participants. Similarly, Figure 11(b) indicates that in general, participants with longer education years are predicted to perform better in tests. Additionally, Figure 11(c) 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 predicted outcomes.

In addition to visualization, there are various methods to assess feature importance in neural network models (Molnar, 2020). A straightforward technique is feature permutation, that is, 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 data set 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 to little additional information, or that they are irrelevant.

Beyond PDPs and feature permutation, more advanced model interpretation techniques are available and widely used in the machine learning community. One such method is SHapley Additive exPlanations (SHAP), which provides consistent and locally accurate attributions for each feature's contribution to a model's prediction (Lundberg & Lee, 2017). SHAP values can offer deeper insight into individual predictions and the global behavior of the model. We encourage readers to explore SHAP and other interpretability tools, such as LIME (Ribeiro et al., 2016) and Integrated Gradients (Sundararajan et al., 2017), especially when applying neural networks in high-stakes or complex research settings.

Discussion

In this article, we presented a step-by-step guide for employing FNNs in psychological research. We started

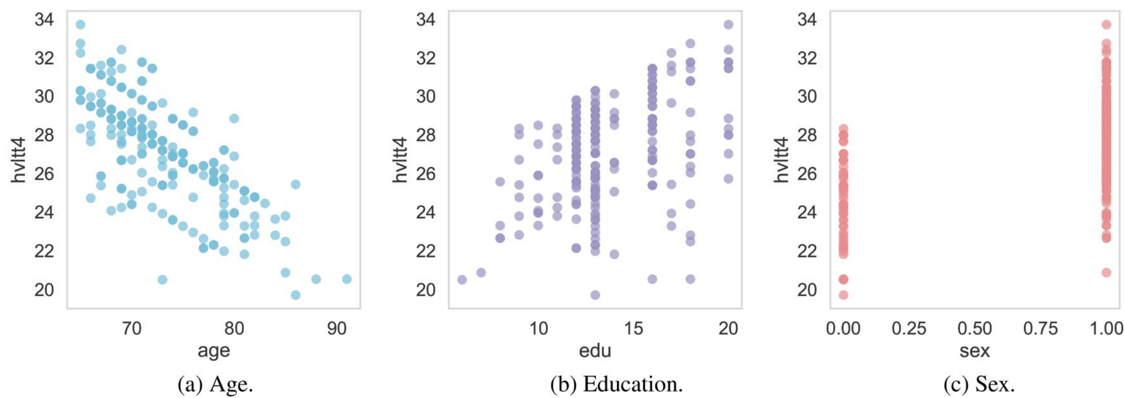


Figure 11. One-way partial dependence plots (PDPs).

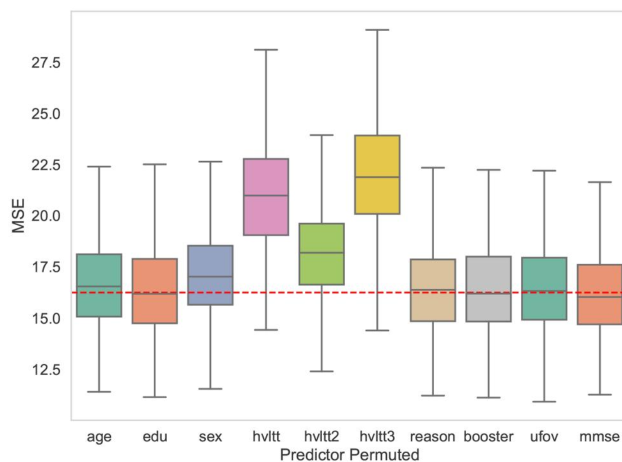


Figure 12. MSE with permuted predictors.

with a concrete example of a basic 3-layer FNN on a psychological data set, 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.

FNNs as a foundation for deep learning

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, making them highly useful for analyzing psychological time-series data, such as predicting suicidal ideation in ecological momentary assessments (EMA; Choo (2025)). However, standard RNNs struggled with vanishing gradients, limiting their ability to model long-range dependencies. To overcome this, Long Short-Term Memory (LSTM) networks (Hochreiter & Schmidhuber, 1997) introduced gating mechanisms to preserve context over extended sequences. Gated Recurrent Units (GRUs; Cho et al. (2014)) further streamlined this approach, offering efficient training on smaller data sets. These models have been applied in areas such as tracking child language development (Portelance et al., 2020; Sagae, 2021) and diagnosing health conditions using data from electroencephalogram (EEG; Cui et al. (2019)) and machine health monitoring systems (MHMS; Zhao et al. (2018)).

CNNs excel in pattern recognition within image data through convolutional layers. Revolutionized spatial data analysis through convolutional layers detect hierarchical patterns in grid-like data, for example, images (LeCun et al., 1989). In psychology, CNNs have automated facial expression analysis, reducing reliance on subjective manual coding in emotion research (Corneanu et al., 2016). They have also applied in neuroimaging, such as predicting brain age from raw MRI data (Cole et al., 2017), identifying biomarkers for conditions traditionally diagnosed through behavioral observation.

Transformers, introduced by Vaswani et al. (2017), disrupted sequential processing paradigms by leveraging self-attention mechanisms to weigh interdependencies across entire data sets. Models like BERT (Devlin et al., 2019) and the GPT series (Brown et al., 2020; Radford et al., 2019) enabled breakthroughs in natural language processing (NLP). For psychologists, these tools facilitate sentiment analysis of free-text

responses (Hoang et al., 2019), personality inference from social media (Peters & Matz, 2024), and even automated therapeutic chatbots (Minerva & Giubilini, 2023). Transformers also support multimodal approaches, such as predicting depressive symptoms from text, audio, video, and sensor data (Lorenzoni et al., 2024; Moura et al., 2023), demonstrating their versatility in capturing nuanced behavioral signals.

Nevertheless, 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.

Comparison with traditional statistical models

FNNs and other machine learning models can offer meaningful advantages over traditional statistical models, such as linear regression, logistic regression, multilevel modeling, and structural equation modeling (SEM), under the certain conditions. Traditional modeling techniques are often theory-driven, interpretable, and effective at hypothesis testing and understanding causal relationships. However, they rely on explicit assumptions (e.g., linearity, homoscedasticity, normality) and predefined model structures, potentially limiting their flexibility when analyzing complex behavioral data.In contrast, neural networks provide greater flexibility and predictive accuracy by automatically detecting complex, nonlinear interactions and patterns without explicit specification. This feature is particularly advantageous for high-dimensional data sets or when dealing with subtle, intricate relationships among variables.

Importantly, machine learning methods and traditional modeling approaches should not be viewed as competing methodologies but rather complementary tools. Integrating nerual network models into psychological research allows researchers to capitalize on their strengths in exploratory analysis and predictive performance, guiding subsequent confirmatory analyses and theory-building with traditional statistical techniques.

Loss functions and learning paradigms

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 article 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, that is, 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).

Article information

Conflict of interest disclosures: Each author signed a form for disclosure of potential conflicts of interest. No authors reported any financial or other conflicts of interest in relation to the work described.

Ethical Principles: The authors affirm having followed professional ethical guidelines in preparing this work. These guidelines include obtaining informed consent from human participants, maintaining ethical treatment and respect for the rights of human or animal participants, and ensuring the privacy of participants and their data, such as ensuring that individual participants cannot be identified in reported results or from publicly available original or archival data.

Funding: This work was supported by Grant R305D210023 from the Institute of Education Sciences, the Lucy Family Institute for Data and Society, and Notre Dame Global.

Role of the Funders/Sponsors: None of the funders or sponsors of this research had any role in the design and conduct of the study; collection, management, analysis, and interpretation of data; preparation, review, or approval of the manuscript; or decision to submit the manuscript for publication.

Acknowledgments: The authors would like to thank Hanna Weber for their comments on prior versions of this manuscript. The ideas and opinions expressed herein are those of the authors alone, and endorsement by the authors' institution or the funding agencies is not intended and should not be inferred.

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*

Abdurahman, S., Vu, H., Zou, W., Ungar, L., & Bhatia, S. (2024). A deep learning approach to personality assessment: Generalizing across items and expanding the reach of survey-based research. *Journal of Personality and Social Psychology*, *126*(2), 312–331. https://doi.org/10.1037/pspp0000480

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

Akiba, T., Sano, S., Yanase, T., Ohta, T., Koyama, M. (2019). Optuna: A next-generation hyperparameter optimization framework. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2623–2631.

Allaire, J., Tang, Y. (2025). Tensorflow: R interface to 'tensorflow' [R package version 2.16.0.9000]. https://github.com/rstudio/tensorflow

Becker, M., Lang, M., Richter, J., Bischl, B., Schalk, D. (2025). Mlr3tuning: Hyperparameter optimization for 'mlr3' [R package version 1.4.0]. https://github.com/mlr-org/mlr3tuning

Bengio, Y. (2012). Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade* (2nd ed., pp. 437–478). Springer.

Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, *13*(1), 281–305.

Bergstra, J., Komer, B., Eliasmith, C., Yamins, D., & Cox, D. D. (2015). Hyperopt: A python library for model selection and hyperparameter optimization. *Computational Science & Discovery*, *8*(1), 014008. https://doi.org/10.1088/1749-4699/8/1/014008

Bishop, C. M., & Nasrabadi, N. M. (2006). *Pattern recognition and machine learning* (Vol. 4) 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. https://doi.org/10.1023/A:1010933404324

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 learn- ers. *Advances in Neural Information Processing Systems*, *33*, 1877–1901.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. arXiv Preprint, arXiv:1406.1078.

Chollet, F., et al. (2015). Keras [Computer software]. https://keras.io

Chollet, F., Allaire, J., et al. (2017). R interface to keras [R package version 1.4.0]. https://github.com/rstudio/keras

Choo, T.-H. J. (2025). *Leveraging recurrent neural networks for predicting suicidal ideation: Advancing the analysis of ecological momentary assessment data*. Columbia University.

Cole, J. H., Poudel, R. P., Tsagkrasoulis, D., Caan, M. W., Steves, C., Spector, T. D., & Montana, G. (2017). Predicting brain age with deep learning from raw imaging data results in a reliable and heritable biomarker. *NeuroImage*, *163*, 115–124. https://doi.org/10.1016/j.neuroimage.2017.07.059

Corneanu, C. A., Simón, M. O., Cohn, J. F., & Guerrero, S. E. (2016). Survey on rgb, 3d, thermal, and multimodal approaches for facial expression recognition: History, trends, and affect-related applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *38*(8), 1548–1568. https://doi.org/10.1109/TPAMI.2016.2515606

Cui, R., Liu, M., & Initiative, A. D. N., Alzheimer's Disease Neuroimaging Initiative (2019). Rnn-based longitudinal analysis for diagnosis of alzheimer's disease. *Computerized Medical Imaging and Graphics: The Official Journal of the Computerized Medical Imaging Society*, *73*, 1–10. https://doi.org/10.1016/j.compmedimag.2019.01.005

Devlin, J., Chang, M.-W., Lee, K., Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: human Language Technologies*, volume *1 (Long and short papers)*, 4171–4186.

Falbel, D., Luraschi, J. (2025). Torch: Tensors and neural networks with 'gpu' acceleration [R package version 0.16.0]. https://torch.mlverse.org/docs

Fritsch, S., Guenther, F., Wright, M. N. (2020). Neuralnet: Training of neural networks [R package version 1.44.6]. https://github.com/bips-hb/neuralnet

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. https://doi.org/10.1007/s12559-023-10179-8

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

Hoang, M., Bihorac, O. A., Rouces, J. (2019). Aspect-based sentiment analysis using bert. *Proceedings of the 22nd Nordic Conference on Computational Linguistics*, 187–196.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, *9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

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. https://doi.org/10.1016/j.ijforecast.2006.03.001

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. https://doi.org/10.3390/healthcare11030285

Jain, A. K. (2010). Data clustering: 50 years beyond k-means. *Pattern Recognition Letters*, *31*(8), 651–666. https://doi.org/10.1016/j.patrec.2009.09.011

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., & Kleinman, K.,. (2001). Active: A cognitive intervention trial to promote independence in older adults. *Controlled Clinical Trials*, *22*(4), 453–479. https://doi.org/10.1016/s0197-2456(01)00139-8

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

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., & Tang, P. T. P. (2016). On large-batch training for deep learning: Generalization gap and sharp minima. arXiv Preprint, arXiv:1609.04836

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.

Kuhn, M. (2025). *Tune: Tidy tuning tools* [R package version 2.0.0.9000]. https://tune.tidymodels.org/tune

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444. https://doi.org/10.1038/nature14539

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. https://doi.org/10.1162/neco.1989.1.4.541

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

Liaw, R., Liang, E., Nishihara, R., Moritz, P., Gonzalez, J. E., & Stoica, I. (2018). Tune: A research platform for distributed model selection and training. arXiv preprint arXiv:1807.05118

Lorenzoni, G., Velmovitsky, P. E., Alencar, P., Cowan, D. (2024). Gpt-4 on clinic depression assessment: An llm-based pilot study. *2024 IEEE International Conference on Big Data (BigData)*, 5043–5049.

Lundberg, S. M., & Lee, S.-I. (2017). A unified approach to interpreting model predictions. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 4768–4777.

Malhotra, A., & Jindal, R. (2022). Deep learning techniques for suicide and depression detection from online social media: A scoping review. *Applied Soft Computing*, *130*, 109713. https://doi.org/10.1016/j.asoc.2022.109713

M., Abadi, A., Agarwal, P., Barham, E., Brevdo, Z., Chen, C., Citro, G. S., Corrado, A., Davis, J., Dean, M., Devin, S., Ghemawat, I., Goodfellow, A., Harp, G., Irving, M., Isard, Jia, Y., Rafal Jozefowicz, L., Kaiser, M., Kudlur, X., Zheng. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems [Software available from tensorflow.org]. https://www.tensorflow.org/

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. https://doi.org/10.1093/bib/bbw068

Minerva, F., & Giubilini, A. (2023). Is ai the future of mental healthcare? *Topoi*, *42*(3), 809–817. https://doi.org/10.1007/s11245-023-09932-3

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

Moura, I., Teles, A., Viana, D., Marques, J., Coutinho, L., & Silva, F. (2023). Digital phenotyping of mental health using multimodal sensing of multiple situations of interest: A systematic literature review. *Journal of Biomedical Informatics*, *138*, 104278. https://doi.org/10.1016/j.jbi.2022.104278

Orrù, G., Monaro, M., Conversano, C., Gemignani, A., & Sartori, G. (2020). Machine learning in psychometrics and psychological research. *Frontiers in Psychology*, *10*, 492685. https://doi.org/10.3389/fpsyg.2019.02970

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. *In Proceedings of the 31st International Conference on Neural Information Processing Systems*, 4768–4777.

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.

Peters, H., & Matz, S. C. (2024). Large language models can infer psychological dispositions of social media users. *PNAS Nexus*, *3*(6), pgae231. https://doi.org/10.1093/pnasnexus/pgae231

Portelance, E., Degen, J., Frank, M. C. (2020). Predicting age of acquisition in early word learning using recurrent neuralnetworks. *Proceedings of the Annual Meeting of the Cognitive Science Society*, 42.

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. https://doi.org/10.1016/j.neucom.2016.12.092

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. https://doi.org/10.1515/itms-2017-0003

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., Gillon, C. J., Hafner, D., Kepecs, A., Kriegeskorte, N., Latham, P., Lindsay, G. W., Miller, K. D., Naud, R., Pack, C. C., … Kording, K. P.,. (2019). A deep learning framework for neuroscience. *Nature Neuroscience*, *22*(11), 1761–1770. https://doi.org/10.1038/s41593-019-0520-2

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), 5393–5397. https://doi.org/10.30534/ijatcse/2020/175942020

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. https://doi.org/10.1038/s42256-019-0048-x

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. https://doi.org/10.1049/el.2018.7980

Sagae, K. (2021). Tracking child language development with neural network language models. *Frontiers in Psychology*, *12*, 674402. https://doi.org/10.3389/fpsyg.2021.674402

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: The Official Journal of the International Neural Network Society*, *61*, 85–117. https://doi.org/10.1016/j.neunet.2014.09.003

Snoek, J., Larochelle, H., & Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. *In* Proceedings of the 26th International Conference on Neural Information Processing Systems, 2951–2959.

Su, C., Xu, Z., Pathak, J., & Wang, F. (2020). Deep learning in mental health outcome research: A scoping review. *Translational Psychiatry*, *10*(1), 116. https://doi.org/10.1038/s41398-020-0780-3

Sundararajan, M., Taly, A., Yan, Q. (2017). Axiomatic attribution for deep networks. *In Proceedings of the 34th International Conference on Machine Learning*, 3319–3328.

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. https://doi.org/10.1002/asi.23596

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. https://doi.org/10.3390/a13010007

Urban, C. J., & Gates, K. M. (2021). Deep learning: A primer for psychologists. *Psychological Methods*, *26*(6), 743–773. https://doi.org/10.1037/met0000374

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser., & Polosukhin, I. (2017). Attention is all you need. *In* Proceedings of the 26th International Conference on Neural Information Processing Systems, *2951–2959*.

Venables, W. N., Ripley, B. D. (2002). *Modern applied statistics with s* (Fourth) [ISBN 0-387-95457-0]. Springer. https://www.stats.ox.ac.uk/pub/MASS4/

Wilson, D. R., & Martinez, T. R. (2003). The general inefficiency of batch training for gradient descent learning. *Neural Networks: The Official Journal of the International Neural Network Society*, *16*(10), 1429–1451. https://doi.org/10.1016/S0893-6080(03)00138-2

Xu, R., & Wunsch, D. (2005). Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, *16*(3), 645–678. https://doi.org/10.1109/TNN.2005.845141

Yarkoni, T., & Westfall, J. (2017). Choosing prediction over explanation in psychology: Lessons from machine learning. *Perspectives on Psychological Science: A Journal of the Association for Psychological Science*, *12*(6), 1100–1122. https://doi.org/10.1177/1745691617693393

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

Zhao, R., Wang, D., Yan, R., Mao, K., Shen, F., & Wang, J. (2018). Machine health monitoring using local feature-based gated recurrent unit networks. *IEEE Transactions on Industrial Electronics*, *65*(2), 1539–1548. https://doi.org/10.1109/TIE.2017.2733438

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