Tutorial: Artificial Neural Networks to Analyze Single-Case Experimental Designs

Marc J. Lanovaz[1,2] and Jordan D. Bailey[3]

[1]École de psychoéducation, Université de Montréal

[2]Centre de recherche de l'Institut universitaire en santé mentale de Montréal

[3]Franciscan Missionaries of Our Lady University

Author Note

This study was supported in part by a salary award from the Fonds de recherche du Québec – Santé (#269462) to the first author. The data and code are available from the Open Science Framework at: https://osf.io/68kgy/. A version of this article was published as a preprint on PsyArXiv: https://psyarxiv.com/3qzhp/ .

Correspondence concerning this article should be addressed to Marc J. Lanovaz, École de psychoéducation, Université de Montréal, C.P. 6128, succursale Centre-Ville, Montreal, QC, Canada, H3C 3J7. Email: marc.lanovaz@umontreal.ca, Phone: 1 514-343-6111 #81774

Abstract

Since the start of the 21st century, few advances have had as far-reaching impact in science as the widespread adoption of artificial neural networks in fields as diverse as fundamental physics, clinical medicine, and psychology. In research methods, one promising area for the adoption of artificial neural networks involves the analysis of single-case experimental designs. Given that these types of networks are not generally part of training in the psychological sciences, the purpose of our paper is to provide a step-by-step introduction to using artificial neural networks to analyze single-case designs. To this end, we trained a new model using data from a Monte Carlo simulation to analyze multiple baseline graphs and compared its outcomes to traditional methods of analysis. In addition to showing that artificial neural networks may produce less error than other methods, this tutorial provides information to facilitate the replication and extension of this line of work to other designs and datasets.

Keywords: artificial intelligence, deep learning, error rate, machine learning, n-of-1 trial, single-case designs

Tutorial: Artificial Neural Networks to Analyze Single-Case Experimental Designs

In 2019, Yoshua Bengio, Geoffrey Hinton, and Yann LeCun received the A. M. Turing prize, often referred to as the Nobel Prize in computing, for revolutionizing artificial neural networks and deep learning (Association for Computing Machinery, 2019). The development of deep learning networks has ushered a new era in artificial intelligence research. Notably, these networks have led to far reaching developments in image classification (Rawat & Wang, 2017), video analysis (Wu et al., 2018), and natural language processing (Young et al., 2018), which have been widely adopted by large technology companies.

In recent years, the application of artificial neural networks has moved into new fields such as healthcare, education, and psychology (Coelho & Silveira, 2017; Kaur & Sharma, 2019; Okubo et al., 2017; Miotto et al., 2018; Pham et al., 2017). The field of medicine has adopted neural networks for decisional support. For example, neural networks are commonly used in oncology for detecting various types of cancer, in orthopedics for diagnosis and prognosis related to skeletal conditions, and in cardiology for predicting risk of heart attack (Haglin et al., 2018). These tools allow clinicians to make decisions relying not only their own learning histories, or contact with the literature, but on demonstrably accurate models. In education, large datasets have allowed researchers to use neural networks to predict student academic performance. For instance, Amrieh et al. (2016) relied upon a large dataset that contained features consisting of student demographics, parent involvement, and in-class behavior to predict student achievement. The authors achieved these results with large amounts of data that could be collected from learning management systems.

Artificial neural networks have also produced an impact on psychological research. In experimental examples of neural networks, some studies have examined the use of artificial neural networks to develop and test models of behavior (Burgos 2003, 2007; Ninness & Ninness, 2020). In applied research examples, Linstead et al. (2015, 2017) have used neural networks to solve predict which children with autism spectrum disorders benefit from early intensive behavioral intervention whereas other researchers have applied neural networks to the measurement of behavior (Dufour et al., 2020; Min & Fetzner, 2019; Rad & Furlanello, 2016). Another fruitful area of research has involved using neural networks to improve the diagnosis of psychological disorders (Kaur & Sharma, 2019). These previous examples show the diversity and potential applications of neural networks in psychological research.

In research methods, one promising area for the adoption of artificial neural networks involves the analysis of single-case experimental designs (Lanovaz et al., 2020). Single-case experimental designs, often referred to as n-of-1 trials in the field of medicine, involve the repeated measurement of a dependent variable in the absence and presence of an independent variable (Smith, 2012). While repeatedly measuring the dependent variable, the experimenter manipulates the presentation of the independent variable in such a way to control for the potential effects of confounding variables (e.g., maturation, history; see Kratochwill et al., 2010 for thorough review of the topic). In the psychological and educational sciences, researchers have widely adopted the use of single-case research in both basic and applied research (Kazdin, 2019; Machalicek & Horner, 2018; Shadish & Sullivan, 2011; Soto, 2020). Moreover, practitioners in many subfields of psychology are strongly encouraged to use single-case designs to monitor and assess the effects of interventions in their practice (Cooper et al., 2019; Kazdin, 2011; Page & Thelwell, 2013). In medicine, several researchers have also advocated for the adoption of single-case design to support the development of personalized and precision medicine (Davidson et al., 2018; Schork, 2015). As such, single-case experimental designs are common research methods that have transformed, and may continue to transform, how we conduct research and practice in diverse fields.

Despite their widespread adoption, one issue with single-case experimental designs is that visual analysis is often described as the gold standard for determining whether an observed change is significant (Manolov & Vannest, 2019). Using visual analysis as a gold standard is a controversial and questionable practice because researchers have found it to be unreliable (Ninci et al., 2015). This reliance on visual analysis may turn away researchers and practitioners who want to adopt a rigorous, empirically supported approach to data analysis.

To address this issue, researchers have proposed using different methods to analyze single-case designs including randomization tests (e.g., Bouwmeester, & Jongerling, 2020; Levin et al., 2018), traditional statistics (e.g., Harrington & Velicer, 2015), and visual aids (e.g., Fisher et al., 2003; Manolov & Vannest, 2019), but none have become well-established and visual analysis remains the norm. Nonetheless, the most studied method is the dual-criteria method, a visual aid (Fisher et al., 2003). The dual-criteria method involves tracing a continuation of the mean and trend lines from baseline phase unto to the treatment phase and examining the number of points that fall above (or below) both lines. If the number of points is equal or higher than a certain threshold (derived from the binomial distribution), the experimenter may conclude that a significant changed has occurred. Although the most studied method, the dual-criteria method has limited power, which remains a serious concern for researchers (Fisher et al., 2003; Manolov & Vannest, 2019; Lanovaz & Turgeon, 2020).

In a recent proof of concept, Lanovaz et al. (2020) found that artificial neural networks produced less decision-making errors and had more power than the dual-criteria method. However, the study limited its analysis to quasi-experimental single-case designs (i.e., AB design). To make these models useful for researchers, developing models to analyze experimental single-case designs (e.g., multiple baseline, changing-criterion, alternating treatment) appears essential. Furthermore, researchers should conduct additional studies on integrating different feature sets as input, using datasets with other distributions, and tuning the hyperparameters during training. One challenge in extending and replicating this line of work is that many researchers and practitioners remain unfamiliar with artificial neural networks, their application to single-case designs, or both. Our tutorial aims to address this barrier by explaining how to develop, train, and test artificial neural networks for analyzing single-case experimental designs.

Artificial Neural Networks

Artificial neural networks consist of neurons and layers interconnected together (Goodfellow et al., 2016). Figure 1 presents the artificial neural network used as part of the current tutorial. As shown in Figure 1, neurons in one layer are generally connected to one or more neurons in another layer, which form a network (a type of directed, weighted graph). The connections involve algorithms (i.e., sets of computer instructions designed to solve problems) that transform the data from one layer to another. During this transformation, the network applies mathematical operations to the input neurons to produce a value for the output neuron.

Artificial neural networks typically consist of three types of layers: an input layer, a hidden layer, and an output layer. The input layer contains the features of the data. The features are the measured characteristics of the data that are used by the network to produce a prediction (Pereira et al., 2009). In single-case designs, the mean of all points in a given phase could represent one continuous feature. Another feature could be categorical such as the data path trend (e.g., downward, upward, flat) for each phase. The network then transforms the data from the input layer prior to passing it to the hidden layer using mathematical operations. This transformation includes an activation function, which allows the modeling of nonlinear patterns. The product of this transformation is a pre-determined number of neurons in the hidden layer. Many approaches exist for determining the number of hidden layers, but these simply serve as an initial architecture for approaching the problem (Panchal et al., 2011). For most simple datasets (as in single-case designs), having one or two hidden layers is generally sufficient to approximate most nonlinear functions (Heaton, 2015). If the artificial neural network contains a single hidden layer, the neurons from the hidden layer are once again transformed by an activation function to produce the output layer.

If a neural network contains no hidden layer and the activation function is sigmoid (often used for binary classification), the network is roughly equivalent to a logistic regression (i.e., a statistical approach designed to predict the value of a categorical variable). If the network contains no hidden layers and has a linear activation function, it is nearly equivalent to linear regression (i.e., a statistical approach designed to predict a dependent variable using a straight line; Perez & Reyes, 2001). Therefore, the presence of a hidden layer or hidden layers makes a significant difference as does the activation function. A deep learning network refers to an artificial neural network that has more than one hidden layer (Deng, 2014). In this case, the algorithm passes the data from one hidden layer to another.

When conducing classification tasks, the output layer contains the class label. A class label is the property that we are interested in predicting or classifying. The network uses the value of class labels to evaluate the degree of error that it produces (Pereira et al., 2009). In single-case analysis, this class label would typically involve whether the graph shows a significant change or not. When training a model, the network uses the input data to predict class label values. The predicted class label values are then compared to the true class label values using a loss function. This computation provides the error for the predictions. Then, the algorithm backpropagates the derivative (i.e., the slope of the tangent) of this error to the previous layers by updating the model. The process then repeats itself, which makes the model more and more precise in estimating the true values of the class labels (Noriega, 2005), which is not unlike a shaping procedure used in learning theory. Each iteration of this process is referred to as an epoch.

For efficiency, the implementation of neural artificial neural networks relies on linear algebra. That is, the computations for calculating the output involve conducting operations on vectors and matrices. Moreover, the backpropagation of the error necessitates calculus to uncover its derivative. Although not essential for developing novel networks, a comprehension of the mathematical foundations underlying them may prevent misapplications. To this end, we present the mathematical foundations of our artificial neural network in Appendix A.

Preliminary Steps

Artificial neural networks require intensive and complex computations that would be challenging for someone without training in mathematics or software engineering to implement. Therefore, software engineers have developed packages in common programming languages to facilitate implementation. Arguably, one of the most popular packages for artificial neural networks is tensorflow for Python, which we will use as part of the current tutorial. The first step is to install a Python environment as well as the necessary packages to run the code. For those unfamiliar with coding in Python, we recommend that you complete the steps described in Appendix B. For those who are familiar, the procedures in the current study were tested using version 3.7.9 of Python, version 1.2.1 of pandas, version 0.23.2 of scikit-learn and version 2.3.0

of tensorflow. Prior to beginning the procedures, you should also download the data and code available in our online repository at: https://osf.io/68kgy/ (Lanovaz, 2021). The lines of code described in the text are available in the "ANN_step-by-step.py" file to facilitate copying and pasting.

Training Dataset

When developing new decision-making models, we need a dataset for which we already have the class labels to train the model. For example, Lanovaz et al. (2020) had two expert visual analysts categorize 1,070 graphs from theses and dissertations as showing a clear change or no clear change. The drawback with using nonsimulated data is that we have no way of knowing the true outcome displayed by the graph. Assume that a visual analyst concludes that a graph shows a clear change. How can one be sure that the graph represents a *true* change in behavior? Maybe the observed change was the result of some uncontrolled variable outside the measurement setting. Validating visual analysis with statistics or multiple raters may increase the believability of the visual analysis of single-case designs, but even then, we can never know with certainty whether the graph displays a true change or not as no method has an accuracy of 100%. The only conclusion that may be drawn is that the raters or the different methods agree or disagree with each other (commonly referred to as correspondence; see Ford et al., 2020; Wolfe et al., 2018).

An alternative to using real-life data is to simulate data using a Monte Carlo simulation. Monte Carlo simulation is a resampling method that involves using a computer to generate inputs from a known distribution (Kroese et al., 2014). In effect, researchers may apply this approach to create experimental data. For example, Fisher et al. (2003) used Monte Carlo simulation to evaluate the dual-criteria method, the visual aid that they designed to support practitioners and researchers in analyzing single-case graphs. The two advantages of Monte Carlo simulation are that (a) the experimenter knows the true values of the class labels in the generated dataset, and (b) it is possible to generate a lot more single-case graphs that could be realistically extracted elsewhere (e.g., 100,000), thus saving much time and effort while still obtaining valid results.

As the most popular single-case design in research is the multiple baseline design (Coon & Rapp, 2018; Shadish & Sullivan, 2011; Smith, 2012), our tutorial will develop a model to analyze such designs. Figure 2 presents a multiple baseline graph. The multiple baseline design involves repeatedly measuring a dependent variable across behaviors, contexts or subjects (i.e., tiers) during a baseline condition (Phase A), and then sequentially introducing the independent variable in each tier (Phase B). More specifically, the experiment introduces the independent variable in a tier only when a change was observed in the prior tier. This manipulation allows for the control of maturational and historical variables. Finally, the analysis involves examining whether changes in the dependent variable coincide with the introduction of the dependent variable in each tier.

Using procedures previously described by Lanovaz and Turgeon (2020), we generated 100,000 multiple baseline graphs with three tiers. For each graph, the code produced three data series with 14 points generated from a normal distribution with a mean of 0 and standard deviation of 1. To remain consistent with Lanovaz and Turgeon, the first tier had 3 points in Phase A and 11 points in Phase B, the second tier had 6 points in Phase A and 8 points in Phase B, and the third tier had 9 points in Phase A and 5 points in Phase B. The algorithms added a constant of 10 to all values to prevent negative values. The final step involved adding a standardized mean difference (SMD) to half of the graphs (i.e., 50,000) to simulate an effect and test for power. Our program randomly selected the SMD value for each graph from a uniform distribution that ranged from 1 to 3. Then, this value was added to the Phase B points across all tiers. This tutorial does not aim to teach how to conduct Monte Carlo simulations, but our code remains available for inspection and replication in the repository (see “generate_mb.py” file for code).

Features

As indicated previously, features function as the input data for the algorithm. Because the phase lengths and measurement scales may differ across multiple baseline designs, we did not use the actual data point values as features. Our code transformed each tier to a z-score. A z-score is a dimensionless measure that represents the number of standard deviations that a point is above or below the mean. This transformation produced graphs that had similar properties: Each graph had mean of 0 and a standard deviation of 1. Computing a z-score involved subtracting the mean of all points in a graph from the value of a specific point and dividing this result by the standardized deviation of all points in the graph. Then, our program extracted eight continuous features for each tier of the multiple baseline graph: 1) the mean of points in Phase A, 2) the mean of points in Phase B, 3) the standard deviation of points in Phase A, 4) the standard deviation of points in Phase B, 5) the intercept of the least squares regression line (LSRL) for Phase A, 6) the slope of LSRL for Phase A, 7) the intercept of LSRL for Phase B, and 8) the slope of LSRL for Phase B. These features represent the level, variability, the immediacy, and trend for each phase.

At this point, the reader should note that these features were selected rather arbitrarily to represent important dimensions of single-case graphs and to remain consistent with Lanovaz et al. (2020) who used the same features for the analysis of AB graphs. That said, researchers and practitioners extending this line of research should strongly consider testing other combinations of variables or selecting a systematic approach to identifying the relevant features (see Cai et al., 2018; Visalakshi & Radha, 2014). The file “TrainingFeatures.csv” shows how the data were organized for our study. You should download the previous file as well as all other .csv data files prior to starting the next steps. In .csv files, the data for each row are entered in different lines whereas commas in a row indicate a change in column. Note that you may open, write and read a .csv file with common spreadsheet programs such as Apple Numbers®, Microsoft Excel® and Google Sheets ®. To load the data, open your integrated development environment and set your working directory to the location of the downloaded files (see Appendix A for help). Run the following lines of code to import the features:

```
1 #Load features
2 import pandas as pd
3 x_train = pd.read_csv("TrainingFeatures.csv", header = None)
4 x_train = x_train.values
5
```

The second line imports a package (pandas) to read .csv files. The third line loads the features and the fourth line transforms the data frame to a matrix, which is necessary for training the artificial neural network. The variable x_train now contains the 24 features for each of the 100,000 graphs. The data in this variable should appear in the Variable Explorer (if you are using an integrated development environment).

Labels

The class label involved a binary variable: change (value = 1) or no change (value = 0). In our example, the label was 0 for the 50,000 graphs showing no effect (i.e., no SMD added) whereas the label was 1 for the 50,000 graphs showing an effect (i.e., SMD added to points of Phase B). You must now load the labels to train the artificial neural network:

```
6 #Load labels
7 y_train = pd.read_csv("TrainingLabels.csv", header = None)
8 y_train = (y_train.values).flatten()
9
```

The previous code imported the labels (line 7) and transformed the structure of the data to a vector (line 8). The y_train variable represents the data labels for the 100,000 graphs in a vector format.

Training the Model

The next step involves building our artificial neural network. Figure 1 presents the artificial neural network tested as part of the current study. The network has 24 input features, 12 hidden neurons and 1 class label. The 12 hidden neurons were fixed at a little less than half the sum of features and labels for each sample (Heaton, 2015). The hidden layer is dense because each neuron is connected with every other neuron in the prior and following layers. When each neuron is connected with all neurons in the previous and following layers, we can refer to the network as a fully-connected artificial neural network or as a dense neural network. To build the network, the first step is to import the necessary packages and methods:

```
10 #Import packages and methods
11 import numpy as np
12 np.random.seed(48151)
13 import tensorflow as tf
14 from tensorflow.keras.layers import Dense
15 from tensorflow.keras.models import Sequential
16 tf.random.set_seed(48151)
17
```

We recommend that you set the random seeds at the same value as us to produce consistent results (see lines 12 and 16).

Next, you create a model by running the following lines of code:

```
18 #Create an artificial neural network
19 ann = Sequential()
20 #Add first hidden layer
21 ann.add(Dense(12, input_shape=(24, ), activation = "relu"))
22 #Output layer
23 ann.add(Dense(1, activation='sigmoid'))
24 #Loss function
25 ann.compile(loss='binary_crossentropy')
```

```
26
```

Line 19 informs Python to create a new neural network, named ann, to which we will add layers. Line 21 plays two roles: (a) it provides information on the number of features that the model should take as input and (b) it adds the hidden layer. In our case, the number 12 represents the number of neurons in the hidden layer, input_shape indicates that we have 24 features organized as a vector, and activation specifies that we are using the relu function. Line 23 provides details regarding the output layer: a single class label with the sigmoid activation function, which we used to remain consistent with Lanovaz et al. (2020). The last line compiles the model and tells it to use the binary cross-entropy function for computing the loss (deviation of the predictions from the true values).

The subsequent step is to train the model with the data by running the following line of code:

```
27 #Train model
28 ann.fit(x_train, y_train, epochs = 20, class_weight = {0:1, 1:0.25})
29
```

These lines of code train the model to recognize the patterns in the data. The results are a model, ann in our example, which we will use to predict the labels on novel untrained features. The epoch parameter represents the number of times the model updated itself (i.e., computed the error and backpropagated the gradient to the previous layers). The class_weight parameter decreases the relative weight of false negatives when compared to the false positives in the computation of the overall error of the model. Manipulating this parameter appears important because researchers are typically more concerned with false positives. In our case, we chose a ration of 1:4 because acceptable values are typically .05 for Type I error rate and .20 for Type II error rate (i.e., power of .80).

Testing the Model

Now that we have a trained model, the next step involves testing its appropriateness. When testing models, you should always use data that were *not* involved in training. In more psychological terms, testing models examines generalization to untrained exemplars. To test for generalization, we generated novel multiple-baseline graphs with different properties than those used during training.

Test Dataset

The test dataset involved a total of 2,000 graphs that were simulated in a manner similar to the training dataset, with the exception of autocorrelation[1]. Of those graphs, 1,000 showed an effect and the remaining 1,000 showed no effect. For each pair of graphs, the algorithm randomly selected an autocorrelation value from a normal distribution with a mean of .30 and a standard deviation of .15 and applied it to the error term, which is between the mean autocorrelations reported in the literature by two studies (Barnard-Brak et al., 2021; Shadish & Sullivan, 2011). This test dataset provides a unique opportunity to check for generalization as the data used for training in the current study had no autocorrelation programmed. To load the test data, you should write and run the following lines of code:

```
#Import test data
x_test = pd.read_csv("TestFeatures.csv", header = None)
y_test = pd.read_csv("TestLabels.csv", header = None)
x_test = x_test.values
y_test = (y_test.values).flatten()

```

In the previous block of code, x_test represents a matrix of the test features whereas y_test represents a vector for the true labels (i.e., ground truth) of the test set.

[1] Autocorrelation represents the correlation between the value of a point and the value of the point (or points) that precedes it.

Predictions

The model may now make predictions by using the x_test matrix. To do so, you should run the following:

```
36 #Get predictionss on test set
37 predictions = np.round(ann.predict(x_test))
38
```

The code ann.predict(x_test) predicts the class labels based on the features in x_test and the ann model that we had previously developed. The predictions need to be rounded to 0 or 1 as the network provides a continuous probability between 0 and 1. The result produces a vector named predictions, which shows the responses of the model to each of the 2,000 graphs.

Outcome Measures

When examining the validity of different methods to analyze single-case graphs, the three main concerns are typically accuracy, Type I error rate, and power (Manolov & Vannest, 2019). Accuracy represents to what extent the predictions agree with the true values. To compute accuracy, you should divide the number of agreements by the number of graphs:

```
39 #Accuracy measure
40 accuracy = np.sum(predictions.flatten() == y_test)/2000
41 print(accuracy)
42
```

The above code computes accuracy (line 40) and then prints it (line 41) in the console. Your console (bottom right panel of the spyder integrated development environment) should show that this accuracy is .9475.

However, accuracy does not provide information on the type of error produced by the model. The next step is thus to calculate Type I error rate. A Type I error, sometimes referred to as a false positive, represents the probability of a model predicting a significant effect when the graph shows no true effect. We can compute it by dividing the number of times that the model predicted an effect in graphs that show no true effect by the total number of graphs showing no true effect:

```
43 #Type I error rate
44 idx_noeffect, = np.where(y_test == 0)
45 typeIerror = np.sum(predictions[idx_noeffect])/len(idx_noeffect)
46 print(typeIerror)
47
```

The idx_noeffect variable identifies the graphs that show no true effect. The following line computes the proportion of graphs with no true effects for which the model erroneously identified an effect. The final line prints Type I error rate, which is .024. Finally, power represents the proportion of true positives that were correctly identified by the model. To compute power, the program must divide the number of times that the model detects an effect in graphs showing a true effect by the total number of graphs showing a true effect:

```
48 #Power
49 idx_effect, = np.where(y_test == 1)
50 power = np.sum(predictions[idx_effect])/len(idx_effect)
51 print(power)
52
```

First, the code identifies graphs that show true effects. Then, line 50 computes this power and the last line prints the power. The console should indicate a power of .919 for our model.

Comparison Measures

Even though the previous values may seem adequate, the predictions of artificial neural networks should be compared to another methods of analysis. Arguably, the most popular method to analyze AB-type designs in the research literature is the dual-criteria method (Fisher et al., 2003). Implementing the dual-criteria method involves drawing a continuation of the mean and trend lines from baseline to treatment and comparing the number of points that fall above (if an increase is expected) or below (if a decrease is expected) both lines with a preset threshold. Our analyses applied the dual-criteria method to each tier of the multiple baseline design individually.

If the number of points falling above both lines met or exceeded the threshold, we categorized the tier as showing a clear change. If the value remained below the threshold, the tier was rated as showing no clear change. When two or more tiers showed a clear change, the whole multiple baseline graph was categorized as showing a significant change. Otherwise, the graph was categorized as nonsignificant. We set the criteria at two or more tiers as prior research has suggested that requiring all three tiers to show a clear change may be overly stringent and produce inadequate power (Lanovaz & Turgeon, 2020; Novotny et al., 2014; Wolfe et al., 2016).

Table 1 (upper section) compares the values obtained using our neural network with those obtained from the dual-criteria method on the simulated test dataset. Clearly, the artificial neural network improved accuracy over the dual-criteria method. The two methods produced similar results for Type I error rate. The main difference lied in power with the artificial neural network producing results with significantly less false negatives. To further test for generalization, we repeated the analysis with the 300 graphs reported in Experiment 2 of Turgeon and Lanovaz (2020). One advantage is that Turgeon and Lanovaz also presented visual analysis data, which is described as a gold standard in the analysis of single-case designs by some researchers (Manolov & Vannest, 2019). The lower half of Table 1 shows the results. Consistently with the initial comparison with simulated graphs, our neural network improved both overall accuracy and power. To replicate the procedures with the Turgeon and Lanovaz data, you may repeat the code described in the Testing the Model section while replacing the “TestFeatures.csv” and

"TestLabels.csv" files by "LanovazandTurgeon(2020)features.csv" and "LanovazandTurgeon(2020)labels.csv", respectively.

Hyperparameter tuning

Our model produced high accuracy on its first run while using parameters similar to those reported by Lanovaz et al. (2020). Models may produce acceptable predictions with default parameters at times, but most cases require hyperparameter tuning in order to achieve desired levels of accuracy. Hyperparameter tuning involves directly manipulating parameters that change the model architecture or learning rate in some way (Yang & Shami, 2020). These hyperparameters are a separate type of parameter from those which are updated as part of the learning process. The idea is to tune these hyperparameters to maximize an accuracy measure or minimize an error measure on a validation set.

Number of Neurons in the Hidden Layer

As part of the tutorial, we will present how to manipulate one common hyperparameter in artificial neural networks: the number of neurons. During hyperparameter tuning, we compare the accuracy (or alternatively the error) of models with different hyperparameter values and keep the model that produces the best outcome. Our network contained 12 neurons in our hidden layer, which represents half the number the number of input neurons. This value is only one of numerous approaches to initially set the number of neurons (Heaton, 2015). To find the optimal number of hidden neurons, the preferred approach involves conducting hyperparameter tuning.

Importantly, we cannot assess the accuracy of varying the number of hidden neurons directly on the training data as the model may produce overfitting (see below for discussion of overfitting). Similarly, it is also unadvisable to use the test data (in our case the Lanovaz and Turgeon dataset) because the model may fail to generalize to novel datasets. To address this issue, experimenters may further subdivide their training data into two sets when conducting hyperparameter tuning: the training set and the validation set. The network uses the training set to train the model (as we have done previously) whereas the validation set is used to check accuracy only. The first step is thus to separate our training data into two sets using the following code:

```
53 #Split training dataset for hyperparameter tuning
54 from sklearn.model_selection import train_test_split
55 x_train, x_valid, y_train, y_valid = train_test_split(x_train, y_train,
56                                          test_size=0.5, random_state=48151)
57
```

Line 54 imports a method to split the dataset in two. The next line splits the original training dataset into two sets: a training set and a validation set. The test_size parameter indicates that 50% of the data should be moved to the validation set and the remaining to the training set. This value was used to have the same amount of data in both sets. You should keep in mind that you can conduct hyperparameter tuning on the test_size parameter in the same manner in which we are currently doing for neurons. For the pedagogical purposes, we focus on a single hyperparameter at a time.

Now that we have a training and a validation set, you should provide a list of the values to tune the models. In our initial example, we set the number of neurons at 12. We can assess whether more or less neurons may produce more accurate models. The following lines produce a list containing the values 3, 6, 9, 12, 15, 18, 21, 24 to test, which will be used to set the number of hidden neurons:

```
58 #List of number of neurons
59 nbneurons_list = range(3,25,3)
60
```

Before assessing the model, we must also initialize the best value of accuracy and the optimal number of neurons associated with it:

```
61 #Initialize best accuracy and the optimal number of neurons at 0
62 best_accuracy = 0
63 optimal_nbneurons = 0
64
```

The next step is to create a loop that will assess the accuracy of the models for each number of neurons:

```
65 #Repeat for each number of neurons value
66 for nbneurons_value in nbneurons_list:
67   #Create model
68   ann = Sequential()
69   ann.add(Dense(nbneurons_value, input_shape=(24, ), activation = "relu"))
70   ann.add(Dense(1, activation='sigmoid'))
71   ann.compile(loss='binary_crossentropy')
72   #Train model
73   ann.fit(x_train, y_train, epochs = 20, class_weight = {0:1, 1:0.25})
74   #Predict result on validation set
75   predictions = np.round(ann.predict(x_valid))
76   #Accuracy on validation set
77   accuracy = np.sum(predictions.flatten() == y_valid)/len(y_valid)
78   #If model improves accuracy, save model, accuracy, and number of neurons
79   if accuracy > best_accuracy:
80     ann.save('best_ann.h5')
81     best_accuracy = accuracy
82     optimal_nbneurons = nbneurons_value
83
```

The code repeats the initial analysis for each number of neurons value in our list using the loop that begins on line 66. You may recognize lines 67 to 77, which are nearly identical to our initial training. The only difference is that accuracy is computed on the validation data rather than the test data (to prevent overfitting). Line 79 compares the accuracy of the current model with the best model. If the new model is more accurate, the code saves the model, the new best_accuracy value and the optimal number of neurons. The model that produces the best accuracy should be saved as best_model.h5 in your working directory. In our example, we chose accuracy as our criterion. Researchers may use other measures such as kappa, Type I error rate or even power to select the best model.

The final step involves examining the predictions produced by our best model using the following code:

```
84 #Check predictions and outcomes on test data from Lanovaz & Turgeon(2020)
85 best_ann = tf.keras.models.load_model('best_ann.h5')
86 predictions = np.round(best_ann.predict(x_test))
87 accuracy = np.sum(predictions.flatten() == y_test)/2000
88 print(accuracy)
89 typeIerror = np.sum(predictions[idx_noeffect])/len(idx_noeffect)
90 print(typeIerror)
91 power = np.sum(predictions[idx_effect])/len(idx_effect)
92 print(power)
93
```

Once again, these lines of code should seem familiar to you. The only differences are that we load the best model (line 85) and we replaced ann by best_ann. The variable optimal_neurons tells us that the best model had 6 neurons. At .947, the accuracy of this model is nearly identical to the one reported in our initial analysis. The Type I error rate and power are both marginally higher in the tuned neural network at .032 and .926, respectively. Thus, conducting hyperparameter tuning on the number of neurons did not improve our predictions. Two hypotheses may explain this observation. First, accuracy was already near its ceiling, so it was difficult to improve. Second, we based our starting hyperparameter values on prior research, which may explain why our values may have been near-optimal from the onset. When using artificial neural networks, this unusual result is an exception rather than the norm. Typically, hyperparameter tuning will improve models considerably, which is particularly true for real-life data with highly variable distributions that may not be as stable as simulated data.

Other Hyperparameters

In the previous example, we tuned a single hyperparameter. With a neural network, other hyperparameters include the number of epochs, number of hidden layers, number of mini-batches, activation function, loss function, learning rate, momentum, and regularization methods (Koutsoukas et al., 2017). You may also test different combinations of hyperparameters simultaneously by embedding multiple loops. Given that we have already coded an example to conduct tuning with one hyperparameter, the focus of the current section will be on describing the other important hyperparameters that experimenters are likely to manipulate and their potential impact on the predictions.

The number of epochs is a central hyperparameter to tune for artificial neural networks. If you use too few epochs, the model may not produce optimal accuracy. Contrarily, having too many epochs will lead to overfitting. Despite being very accurate on the actual data, the models will struggle to generalize to untrained data. Hyperparameter tuning can thus support experimenters in identifying this optimal value[2]. A second hyperparameter to tune is the number of hidden layers. Our example contained a single hidden layer. Increasing the number of hidden layers may allow the model to learn more complex relationships in data (Goodfellow et al., 2019). Adding a layer would simply involve repeating line 21 of our code while leaving out the input_shape parameter. As indicated earlier, deep learning networks have two or more hidden layers.

To improve the computational efficiency of training, tensorflow uses mini batches to train models. During mini batch training, the network updates the model following a small number of

[2]A more efficient way to tune the number of epochs in tensorflow than using a loop involves the ModelCheckpoint method (see https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ModelCheckpoint). When using ModelCheckpoint, the network will run through the epochs only once and keep the model with the highest accuracy or lowest error (depending on parameters set) without running a loop.

samples instead of the entire sample (100,000 samples in our example). In the current tutorial, tensorflow used the default mini batch size of 32 samples. In addition to improving efficiency, using mini batches may facilitate convergence and reduce training time. Different sizes of mini batches may produce different results, which is why this hyperparameter is a common target for tuning. As noted earlier, the activation function is responsible for modeling more complex, nonlinear relationships in data. Figure 3 shows example of three popular activation functions that may be used with binary outcomes. To remain consistent with Lanovaz et al. (2020), we used the relu function (rectified linear unit) for the hidden layer and the sigmoid function for the output. Conducting hyperparameter tuning by testing different activation functions may have further improved the predictions.

Another important hyperparameter for tuning models is the loss function. The loss function computes to what extent the predictions deviate from the true values. The network uses this loss to update the value during backpropagation of the gradient of the error. For training the model in the current tutorial, we used the binomial cross-entropy function, which is based on the logarithmic function. Other relevant options for binary classification that could be further tested during hyperparameter tuning include the hinge loss and squared-hinge loss functions. Finally, the experimenter may manipulate learning, momentum, and regularization. These hyperparameters influence how fast the model will converge (or learn) and promote generalization to untrained samples. Learning rate, momentum and some regularization parameters are regularly packaged into optimizers. The default optimizer to set these hyperparameters in tensorflow is called RMSprop. Alternatives include the adam, adagrad, and sgd optimizers. Reviewing the mathematical differences between these algorithms falls outside the purpose of the current tutorial, but we recommend that you consult Luo et al. (2019) if you want more details about how each algorithm operates.

Automated Tuning

So far, we have focused on writing code manually to tune the hyperparameters. This step-by-step approach was designed to facilitate the understanding of the logic underlying hyperparameter tuning. A second option for tuning hyperparameters that does not require as much in-depth knowledge is automated hyperparameter tuning. Researchers can tune hyperparameters automatically using a package known as AutoKeras. This approach may be especially helpful when researchers do not know which set of hyperparameters to start with. To use this package, we direct readers to use the same approach as in the current paper and then consult the documentation on how to use AutoKeras on tabular (structured) data at: https://autokeras.com.

Application

The ultimate purpose of developing novel neural networks is for researchers to apply them to empirical datasets. To explain how to apply a previously trained model, let's assume that we want to assess whether the multiple-baseline graph depicted on Figure 2 displays an effect or not using our neural network. First, this procedure involves organizing the data in a spreadsheet (e.g., Google Sheets, Microsoft Excel, Apple Numbers). We recommend the spreadsheet format because spreadsheets are a common tool used by researchers to graph single-case data (Dixon et al., 2009; Lehardy et al., 2021). The spreadsheet should contain two columns with no header (see "application.xlsx" for an example). The first column includes the phase labels and the tier number in order. For example, "A1" represents a point associated with the first tier of Phase A whereas "B2" represents a point from Phase B in the second tier. The second column contains the corresponding values for each point (i.e., the dependent variable). Note that we could also transpose the table so that the first row includes all the phase labels and the second row the point values.

As indicated earlier, our model takes 8 features per tier as input (for a total of 24 features): the mean of Phase A, the mean of Phase B, the standard deviation of Phase A, the standard deviation of Phase B, the intercept and slope of Phase A, and the intercept and slope of Phase B. To facilitate implementation, we wrote a function that extract these features automatically. You should download and save the “extract_features.py” and “application.xlsx” files in your working directory, and then run the following code:

```
94 #Import data for a graph
95 series = (pd.read_excel('application.xlsx', header = None)).values
96 #Extract the features used by the program
97 from extract_features import extractFeatures
98 features = extractFeatures(series)
99
```

Line 95 imports the data from the .xlsx file and transforms it to matrix. If your data contains variables names, you should set the header parameter as True. The code then instructs Python to import the function extractFeatures (line 97) and then extracts the features (line 98). The result is a vector, named features, with 24 values.

The final step involves loading the model and applying it to the features extracted from our graph.

```
100 #Import model
101 best_ann = tf.keras.models.load_model('best_ann.h5')
102 #Make prediction on graph
103 prediction = np.round(best_ann.predict(features.reshape(1,-1)))
104 print(prediction)
```

When conducting hyperparameter tuning, we saved our best neural network for future use (see line 80). Line 101 loads this model and labels it, best_ann. If you have been running the entire code from the beginning, the model should already be loaded (see line 85). Finally, line 103 produces a prediction for this graph and line 104 prints the result. A value of 0 indicates no effect whereas a value 1 suggests an effect of the independent variable. In our example, the value is 1, indicating that the graphs show a clear change. This output value appears consistent with the patterns observed on Figure 2. For clarity, consistency and brevity, we presented the application of the neural network using a Python integrated development environment. Alternatively, researchers may develop user-friendly web or mobile apps with a Python framework (e.g., Kivy, Django, Flask) to share their artificial neural networks.

Other Considerations

Overfitting and Underfitting

When training novel models with artificial neural networks, experimenters should remain wary of two important phenomena: overfitting and underfitting. Overfitting relates to the concept of variance in machine learning, which represents the variability of the predictions when compared to the true values. Models with high variance may learn from irrelevant data (i.e., noise) within a training set. The model will then fail to generalize well to novel data (Mutasa et al., 2020). In contrast, underfitting occurs when the model fails to learn from important features. Underfitting relates to bias, which represents the difference between the predictions and the true values. A model that has a high degree of bias will be less complex and fit the training data less closely than a high-variance model (Hastie et al., 2016). A high bias model may fail to learn the relationship between important features (i.e., signal) and outputs within the dataset. Collectively bias and variance are two sources of error that scientists attempt to minimize during training and validation. Minimizing both can be difficult which results in a trade-off between the two. Hyperparameter tuning also affects the degree to which a model may overfit or underfit the data, which is why a validation set is needed during training.

Cross-Validation

In the tutorial, we used a separate dataset from Lanovaz and Turgeon (2020) to examine the generalizability of our model. When researchers have insufficient data or access to a single dataset, an alternative is cross-validation. Cross-validation involves resampling a single dataset into two or more approximately equal-sized sets often referred to as folds and denoted by the letter *k*. At any one time, the algorithm should use a single fold to test or validate the model whereas the remaining folds train the model (Witten et al., 2017). When *k* is equal to two, one-fold is used to train the model and the other is used to test for generalization. When *k* is greater than two, the algorithm should train the model on *k*-1 folds and test for generalization on the remaining fold (i.e., holdout fold).

For example, a design with a *k* equal to three would involve using two folds to train the model and the third fold to test for generalization. Next, the cross-validation should repeat the procedures twice so that each fold is in the test fold exactly once. Importantly, the model is never tested on data on which it has been trained, meaning a new model is created each time a new fold is used for testing (Mahmood & Khan, 2009). Researchers commonly set the number of folds (*k*) at two, five, or ten (Rodriguez et al., 2010). Cross-validation returns an average score across the number of folds, which provides the user with an estimate of the generalizability of the model. In extreme cases, the number of folds can be equal to the number of samples in the set. This approach is known as leave-one-out cross-validation, and although quite computationally expensive, it is useful for small datasets because it provides a score with lower levels of errors (i.e., difference between estimated and true accuracy) when compared with other values of *k* (Wong, 2015).

Types of Networks

Researchers may explore other types of neural networks to train models to analyze single-case graphs. A type of neural network known as a convolutional neural network (CNN) is especially effective at classification tasks involving images. For example, researchers have used CNNs in medicine to classify or detect anomalies in X-rays, MRIs, and other medical tests that produce an image of some kind (Anwar et al., 2018). Given the visual complexity of some of these images, using CNNs may accomplish similar gains in the analysis of single-case graphs. In other words, the CNN may use the image of a graph, rather than the dimensions of the data, as input features. Such an approach would allow practitioners to simply upload an image of their figure and receive the output.

Another type of network that may be of interest for the analysis of single-case graphs is the recurrent neural network (RNN). In RNN, the algorithm analyzes the series of features sequentially rather than concurrently. That is, the order in which features appears becomes important to the analysis. In experimental psychology, Donahoe and Palmer (2004) suggest using as similar methodology as an exercise in interpretation, as some complex behavior may be impossible or difficult to access experimentally. For the analysis of single case designs, RNNs could use the values of the actual points as features instead of using aggregated data (e.g., mean, standard deviation, slope). Using RNN would also have the advantage of dealing with the problem of autocorrelation effectively, which is commonly present in single-case graphs (Shadish and Sullivan, 2011).

A third type of network relevant for single-case designs is the generative adversarial network (GAN; Goodfellow et al., 2014). A GAN consists of two main components: the generator model and the discriminator model. The generator model produces novels samples based on prior data whereas the discriminator model attempts to determine whether these novel samples are real or fake. This competition between the generator and the discriminator allows for the development of realistic samples. Experimenters could thus use GANs to derive new samples of graphs from published data, which would facilitate replication of the current work with data that show more naturalistic patterns. In other words, GANs could replace or supplement Monte Carlo simulation to generate novel data series on which to train or test models derived from artificial neural networks.

Conclusion

Although we provided instructions for classifying the presence or absence of an effect using multiple baseline designs, researchers may use this tutorial to extend our work to ABAB, multielement, or changing-criterion designs. Each type of design would require its own model as patterns and methods of analysis differ across designs. For example, researchers do not deal with trends across conditions in multielement designs in the same manner as they do in ABAB design (Hains & Baer, 1989). Moreover, researchers should explore other combinations of features to improve upon the models proposed in this study and in Lanovaz et al. (2020). In addition to selecting different features, researchers should test the effects of other neural network architectures with varying combinations of hyperparameters to further improve the accuracy of the models. Generating the data with novel methods or including confounding effects in the simulated data would also extend this line of work.

The current tutorial clearly shows that artificial neural networks have the potential to outperform traditional methods of analysis for single-case experimental designs. That said, our model was not compared to other statistical procedures and is not ready to be used as a standalone method. For now, we recommend it as a complement to other methods, especially when patterns remain unclear. Our results show the potential utility of neural networks in the analysis of single-case graphs and aims to spur further research with this approach that has been revolutionizing other fields. Before widespread adoption of artificial neural networks to analyze single-case experimental designs may occur, more research by independent teams using novel datasets is paramount to examine the replicability of the results presented in this tutorial.

References

Amrieh, E. A., Hamtini, T., & Aljarah, I. (2016). Mining educational data to predict student's academic performance using ensemble methods. *International Journal of Database Theory and Application*, *9*(8), 119–136. https://doi.org/10.14257/ijdta.2016.9.8.13

Anwar, S. M., Majid, M., Qayyum, A., Awais, M., Alnowami, M., & Khan, M. K. (2018). Medical image analysis using convolutional neural networks: A review. *Journal of Medical Systems, 42*(11). https://doi.org/10.1007/s10916-018-1088-1

Association for Computing Machinery. (2019, March 27). *Fathers of the deep learning revolution receive ACM A.M. Turing Award*. https://awards.acm.org/about/2018-turing

Barnard-Brak, L., Watkins, L., & Richman, D. M. (2021). Autocorrelation and estimates of treatment effect size for single-case experimental design data. *Behavioral Interventions*, *36*(3), 595-605. https://doi.org/10.1002/bin.1783

Bouwmeester, S., & Jongerling, J. (2020). Power of a randomization test in a single case multiple baseline AB design. *PloS One*, *15*(2), e0228355. https://doi.org/10.1371/journal.pone.0228355

Burgos, J. E. (2003). Theoretical note: Simulating latent inhibition with selection ANNs. *Behavioural Processes,62*(1-3), 183-192. https://doi.org/10.1016/s0376-6357(03)00025-1

Burgos, J. E. (2007). Autoshaping and automaintenance: A neural-network approach. *Journal of the Experimental Analysis of Behavior,88*(1), 115-130. https://doi.org/10.1901/jeab.2007.75-04

Cai, J., Luo, J., Wang, S., & Yang, S. (2018). Feature selection in machine learning: A new perspective. *Neurocomputing, 300*, 70-79. https://doi.org/10.1016/j.neucom.2017.11.077

Coelho, O. B., & Silveira, I. (2017, October). Deep learning applied to learning analytics and educational data mining: A systematic literature review. In *Brazilian Symposium on Computers in Education,* 28(1), 143-152. https://doi.org/10.5753/cbie.sbie.2017.143

Coon, J. C., & Rapp, J. T. (2018). Application of multiple baseline designs in behavior analytic research: Evidence for the influence of new guidelines. *Behavioral Interventions, 33*(2), 160-172. https://doi.org/10.1002/bin.1510

Davidson, K. W., Cheung, Y. K., McGinn, T., & Wang, Y. C. (2018). Expanding the role of N-of-1 trials in the precision medicine era: Action priorities and practical considerations. *NAM Perspectives.* https://doi.org/10.31478/201812d

Deng, L. (2014). A tutorial survey of architectures, algorithms, and applications for deep learning. *APSIPA Transactions on Signal and Information Processing, 3*, E2. Doi:10.1017/atsip.2013.9

Dixon, M. R., Jackson, J. W., Small, S. L., Horner-King, M. J., Lik, N. M. K., Garcia, Y., & Rosales, R. (2009). Creating single-subject design graphs in Microsoft Excel™ 2007. *Journal of Applied Behavior Analysis*, *42*(2), 277-293. https://doi.org/10.1901/jaba.2009.42-277

Donahoe, J. W., & Palmer, D. C. (2004). *Learning and complex behavior*. Ledgetop Publishing.

Dufour, M.-M., Lanovaz, M. J., Cardinal, P. (2020). Artificial intelligence for the measurement of vocal stereotypy. *Journal of Experimental Analysis of Behavior*, *114*(3), 368-380. https://doi.org/10.1002/jeab.636

Fisher, W. W., Kelley, M. E., & Lomas, J. E. (2003). Visual aids and structured criteria for improving visual inspection and interpretation of single-case designs. *Journal of Applied Behavior Analysis, 36*(3), 387-406. https://doi.org/10.1901/jaba.2003.36-387

Ford, A. L., Rudolph, B. N., Pennington, B., & Byiers, B. J. (2020). An exploration of the interrater agreement of visual analysis with and without context. *Journal of Applied Behavior Analysis*, *53*(1), 572-583. https://doi.org/10.1002/jaba.560

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

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2014). *Generative adversarial networks*. ArXiv Preprint. https://arxiv.org/abs/1406.2661

Haglin, J. M., Jimenez, G., & Eltorai, A. E. M. (2018). Artificial neural networks in medicine. *Health and Technology*, *9*(1), 1–6. https://doi.org/10.1007/s12553-018-0244-4

Hains, A. H., & Baer, D. M. (1989). Interaction effects in multielement designs: inevitable, desirable, and ignorable. *Journal of Applied Behavior Analysis*, *22*(1), 57–69. https://doi.org/10.1901/jaba.1989.22-57

Harrington, M., & Velicer, W. F. (2015). Comparing visual and statistical analysis in single-case studies using published studies. *Multivariate Behavioral Research*, *50*(2), 162-183. https://doi.org/10.1080/00273171.2014.973989

Hastie, T., Friedman, J., & Tisbshirani, R. (2016). *The Elements of statistical learning: data mining, inference, and prediction*. Springer.

Heaton, J. (2015). *Artificial intelligence for humans, volume 3: Deep learning and neural networks*. Heaton Research Inc.

Kaur, P., & Sharma, M. (2019). Diagnosis of human psychological disorders using supervised learning and nature-inspired computing techniques: a meta-analysis. *Journal of Medical Systems*, *43*(7), 1-30. https://doi.org/10.1007/s10916-019-1341-2

Kazdin, A. (2019). Single case and idiographic research: Introduction to the special issue. *Behaviour Research and Therapy*, *117*, 1-2. https://doi.org/10.1016/j.brat.2019.03.007

Koutsoukas, A., Monaghan, K. J., Li, X., & Huan, J. (2017). Deep-learning: investigating deep neural networks hyper-parameters and comparison of performance to shallow methods for modeling bioactivity data. *Journal of Cheminformatics, 9*(1). https://doi.org/10.1186/s13321-017-0226-y

Kroese, D. P., Brereton, T., Taimre, T., & Botev, Z. I. (2014). Why the Monte Carlo method is so important today. *Wiley Interdisciplinary Reviews: Computational Statistics*, *6*(6), 386–392. https://doi.org/10.1002/wics.1314

Lanovaz, M. J. (2021). *Data and code for "Tutorial: Artificial neural networks to analyze single-case experimental designs"* [Data set and code]. Open Science Framework. https://osf.io/68kgy

Lanovaz, M. J., & Bailey, J. D. (2021). *Tutorial: Artificial neural networks to analyze single-case experimental designs*. PsyArXiv. https://doi.org/10.31234/osf.io/3qzhp

Lanovaz, M. J., Giannakakos, A. R., & Destras, O. (2020). Machine learning to analyze single-case data: A proof of concept. *Perspectives on Behavior Science.* Advance online publication. https://doi.org/10.1007/s40614-020-00244-0

Lanovaz, M. J. & Turgeon, S. (2020). How many tiers do we need? Type I errors and power in multiple baseline designs. *Perspectives on Behavior Science, 43*(3), 605-616. https://doi.org/10.1007/s40614-020-00263-x

Lehardy, R. K., Luczynski, K. C., Hood, S. A., & McKeown, C. A. (2021). Remote teaching of publication-quality, single-case graphs in Microsoft Excel. *Journal of Applied Behavior Analysis, 54*(3), 1265-1280. https://doi.org/10.1002/jaba.805

Levin, J. R., Ferron, J. M., & Gafurov, B. S. (2018). Comparison of randomization-test procedures for single-case multiple-baseline designs. *Developmental Neurorehabilitation*, *21*(5), 290-311. https://doi.org/10.1080/17518423.2016.1197708

Linstead, E., Dixon, D. R., French, R., Granpeesheh, D., Adams, H., German, R., … & Kornack, J. (2017). Intensity and learning outcomes in the treatment of children with autism spectrum disorder. *Behavior Modification*, *41*(2), 229-252. https://doi.org/10.1177/0145445516667059

Linstead, E., German, R., Dixon, D., Granpeesheh, D., Novack, M., & Powell, A. (2015, December). An application of neural networks to predicting mastery of learning outcomes in the treatment of autism spectrum disorder. In *2015 IEEE 14th International Conference on Machine Learning and Applications* (pp. 414-418). IEEE. https://doi.org/10.1109/ICMLA.2015.214

Luo, L., Xiong, Y., Liu, Y., & Sun, X. (2019, May). *Adaptive gradient methods with dynamic bound of learning rate*. Paper presented at the International Conference on Learning Representations. https://openreview.net/forum?id=Bkg3g2R9FX

Machalicek, W., & Horner, R. H. (2018). Special issue on advances in single-case research design and analysis. *Developmental Neurorehabilitation, 21*(4), 209-211. https://doi.org/10.1080/17518423.2018.1468600

Mahmood, Z., & Khan, S. (2009). On the use of k-fold cross-validation to choose cutoff values and assess the performance of predictive models in stepwise regression. *The International Journal of Biostatistics, 5*(1). https://doi.org/10.2202/1557-4679.1105

Manolov, R., & Vannest, K. J. (2019). A visual aid and objective rule encompassing the data features of visual analysis. *Behavior Modification*. Advanced online publication. https://doi.org/10.1177/0145445519854323

Min, C. H., & Fetzner, J. (2019). Training a neural network for vocal stereotypy detection. *2019 Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, 5451-5455. https://doi.org/10.1109/EMBC.2019.8856626

Miotto, R., Wang, F., Wang, S., Jiang, X., & Dudley, J. T. (2018). Deep learning for healthcare: Review, opportunities and challenges. *Briefings in Bioinformatics, 19*(6), 1236-1246. https://doi.org/10.1093/bib/bbx044

Mutasa, Simukayi, Sun, Shawn, & Ha, Richard. (2020). Understanding artificial intelligence based radiology studies: What is overfitting? *Clinical Imaging, 65*, 96-99. https://doi.org/10.1016/j.clinimag.2020.04.025

Ninness, C., & Ninness, S. K. (2020). Emergent virtual analytics: Modeling contextual control of derived stimulus relations. *Behavior and Social Issues*. Advance online publication. https://doi.org/10.1007/s42822-020-00032-0

Noriega, L. (2005). Multilayer perceptron tutorial. *School of Computing. Staffordshire University*.

Novotny, M. A., Sharp, K. J., Rapp, J. T., Jelinski, J. D., Lood, E. A., Steffes, A. K., & Ma, M. (2014). False positives with visual analysis for nonconcurrent multiple baseline designs and ABAB designs: Preliminary findings. *Research in Autism Spectrum Disorders, 8*(8), 933-943. https://doi.org/10.1016/j.rasd.2014.04.009

Okubo, F., Yamashita, T., Shimada, A., & Ogata, H. (2017, March). A neural network approach for students’ performance prediction. In *Proceedings of the Seventh International Learning Analytics & Knowledge Conference* (pp. 598-599).

Page, J., & Thelwell, R. (2013). The value of social validation in single-case methods in sport and exercise psychology. *Journal of Applied Sport Psychology*, *25*(1), 61-71. https://doi.org/10.1080/10413200.2012.663859

Panchal, G., Ganatra, A., Kosta, Y. P., & Panchal, D. (2011). Behaviour analysis of multilayer perceptrons with multiple hidden neurons and hidden layers. *International Journal of Computer Theory and Engineering, 3*(2), 332-337. https://doi.org/10.7763/ijcte.2011.v3.328

Pereira, F., Mitchell, T., & Botvinick, M. (2009). Machine learning classifiers and fMRI: A tutorial overview. *NeuroImage*, *45*(1). https://doi.org/10.1016/j.neuroimage.2008.11.007

Perez, P., & Reyes, J. (2001). Prediction of particulate air pollution using neural techniques. *Neural Computing & Applications, 10*(2), 165–171. https://doi.org/10.1007/s005210170008

Pham, T., Tran, T., Phung, D., & Venkatesh, S. (2017). Predicting healthcare trajectories from medical records: A deep learning approach. *Journal of Biomedical Informatics*, *69,* 218-229. https://doi.org/10.1016/j.jbi.2017.04.001

Rad, N. M., & Furlanello, C. (2016). Applying deep learning to stereotypical motor movement detection in autism spectrum disorders. *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*, 1235-1242. https://doi.org/10.1109/ICDMW.2016.0178

Rawat, W., & Wang, Z. (2017). Deep convolutional neural networks for image classification: A comprehensive review. *Neural Computation*, *29*(9), 2352-2449. https://doi.org/10.1162/neco_a_00990

Rodriguez, J. D., Perez, A., & Lozano, J. A. (2010). Sensitivity analysis of k-fold cross validation in prediction error estimation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *32*(3), 569–575. https://doi.org/10.1109/tpami.2009.187

Schork, N. J. (2015). Personalized medicine: Time for one-person trials. *Nature, 520*(7549), 609-611. https://doi.org/10.1038/520609a

Shadish, W. R., & Sullivan, K. J. (2011). Characteristics of single-case designs used to assess intervention effects in 2008. *Behavior Research Methods*, *43*(4), 971-980. https://doi.org/10.3758/s13428-014-0516-5

Smith, J. D. (2012). Single-case experimental designs: A systematic review of published research and current standards. *Psychological Methods*, *17*(4), 510-550. https://doi.org/10.1037/a0029312

Soto, P. L. (2020). Single-case experimental designs for behavioral neuroscience. *Journal of the Experimental Analysis of Behavior*, *114*(3), 447-467. https://doi.org/10.1002/jeab.633

Tarlow, K. R., & Brossart, D. F. (2018). A comprehensive method of single-case data analysis: Interrupted Time-Series Simulation (ITSSIM). *School Psychology Quarterly*, *33*(4), 590-603. http://doi.org/10.1037/spq0000273

Visalakshi, S., & Radha, V. (2014). A literature review of feature selection techniques and applications: Review of feature selection in data mining. *2014 IEEE International Conference on Computational Intelligence and Computing Research*, 1-6. https://doi.org/10.1109/ICCIC.2014.7238499

Witten, I. H., Frank, E., Hall, M. A., & Pal, C. J. (2017). *Data mining: Practical machine learning tools and techniques* (4th ed.). Elsevier.

Wolfe, K., Seaman, M. A., & Drasgow, E. (2016). Interrater agreement on the visual analysis of individual tiers and functional relations in multiple baseline designs. *Behavior Modification*, *40*(6), 852-873. https://doi.org/10.1177/0145445516644699

Wolfe, K., Seaman, M. A., Drasgow, E., & Sherlock, P. (2018). An evaluation of the agreement between the conservative dual-criterion method and expert visual analysis. *Journal of Applied Behavior Analysis*, *51*(2), 345-351.

Wong, T.-T. (2015). Performance evaluation of classification algorithms by k-fold and leave-one-out cross validation. *Pattern Recognition*, *48*(9), 2839-2846. https://doi.org/10.1016/j.patcog.2015.03.009

Wu, D., Sharma, N., & Blumenstein, M. (2017, May). Recent advances in video-based human action recognition using deep learning: a review. In *2017 International Joint Conference on Neural Networks* (pp. 2865-2872). IEEE.

Yang, L., & Shami, A. (2020). On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing*, *415*, 295–316. https://doi.org/10.1016/j.neucom.2020.07.061

Young, T., Hazarika, D., Poria, S., & Cambria, E. (2018). Recent trends in deep learning based natural language processing. *IEEE Computational Intelligence magazine*, *13*(3), 55-75. https://doi.org/10.1109/MCI.2018.2840738

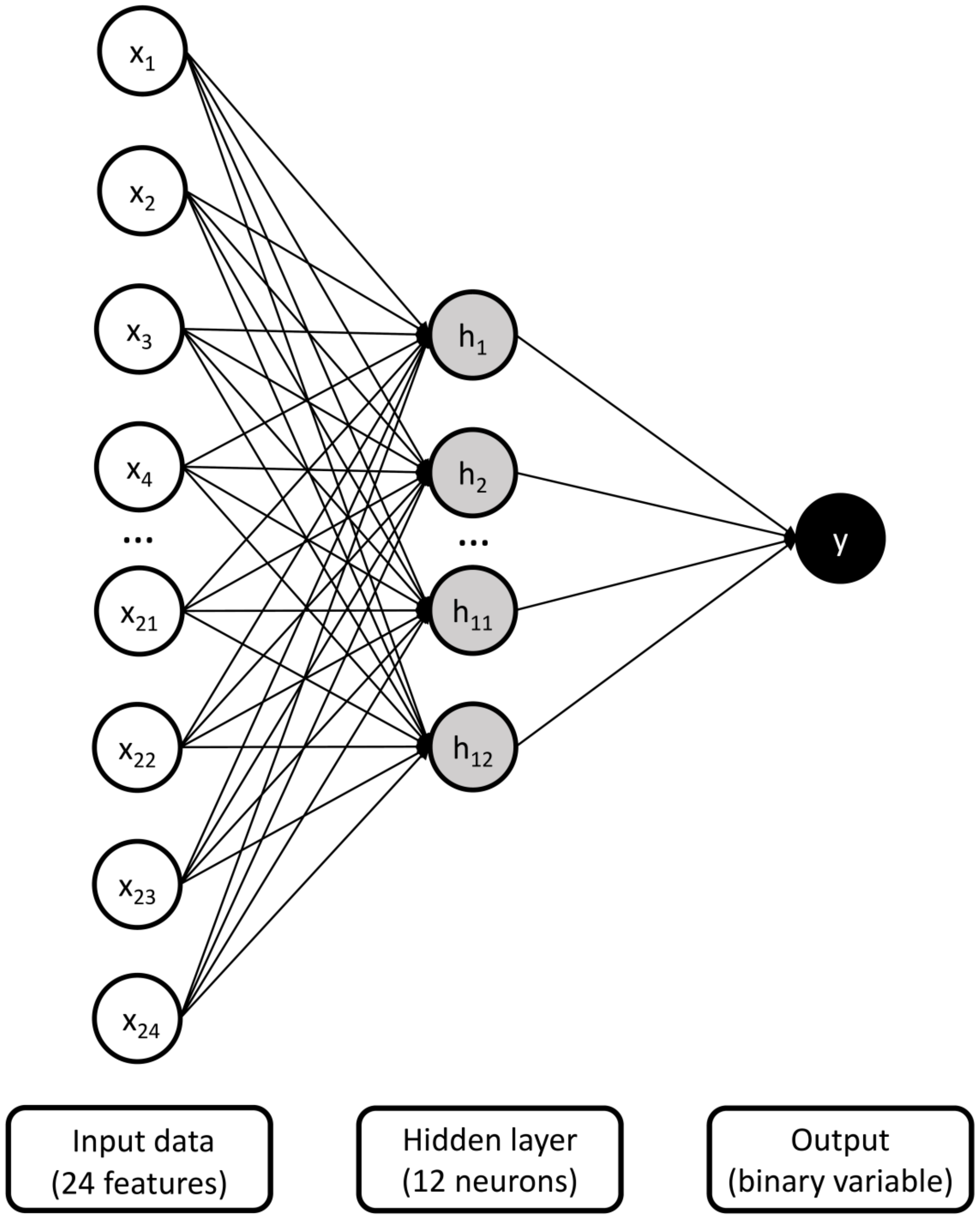
Table 1

Accuracy, Type I Error Rate, and Power of Different Methods of Analysis

	Measure		
	Accuracy	Type I Error Rate	Power
Simulated Test Graphs (*n* = 2,000)			
Artificial Neural Network	.948	.024	.919
Dual-Criteria Method	.819	.025	.663
Lanovaz and Turgeon (2020) Graphs (*n* = 300)			
Artificial Neural Network	.977	.013	.967
Visual Analysts (Mean)	.913	.007	.833
Dual-Criteria Method	.900	.013	.813

Figure 1

Visual Representation of the Design of the Artificial Neural Network in the Tutorial



Note. The features represent the mean, standard deviation, intercept and slope of the standardized data for each tier in the multiple baseline graph (3 tiers x 8 features). The output is a whether the graph shows a clear change (value = 1) or not (value = 0).

Figure 2

Example of a Multiple Baseline Design with Three Tiers (i.e., Conditions, Behaviors, or Participants)

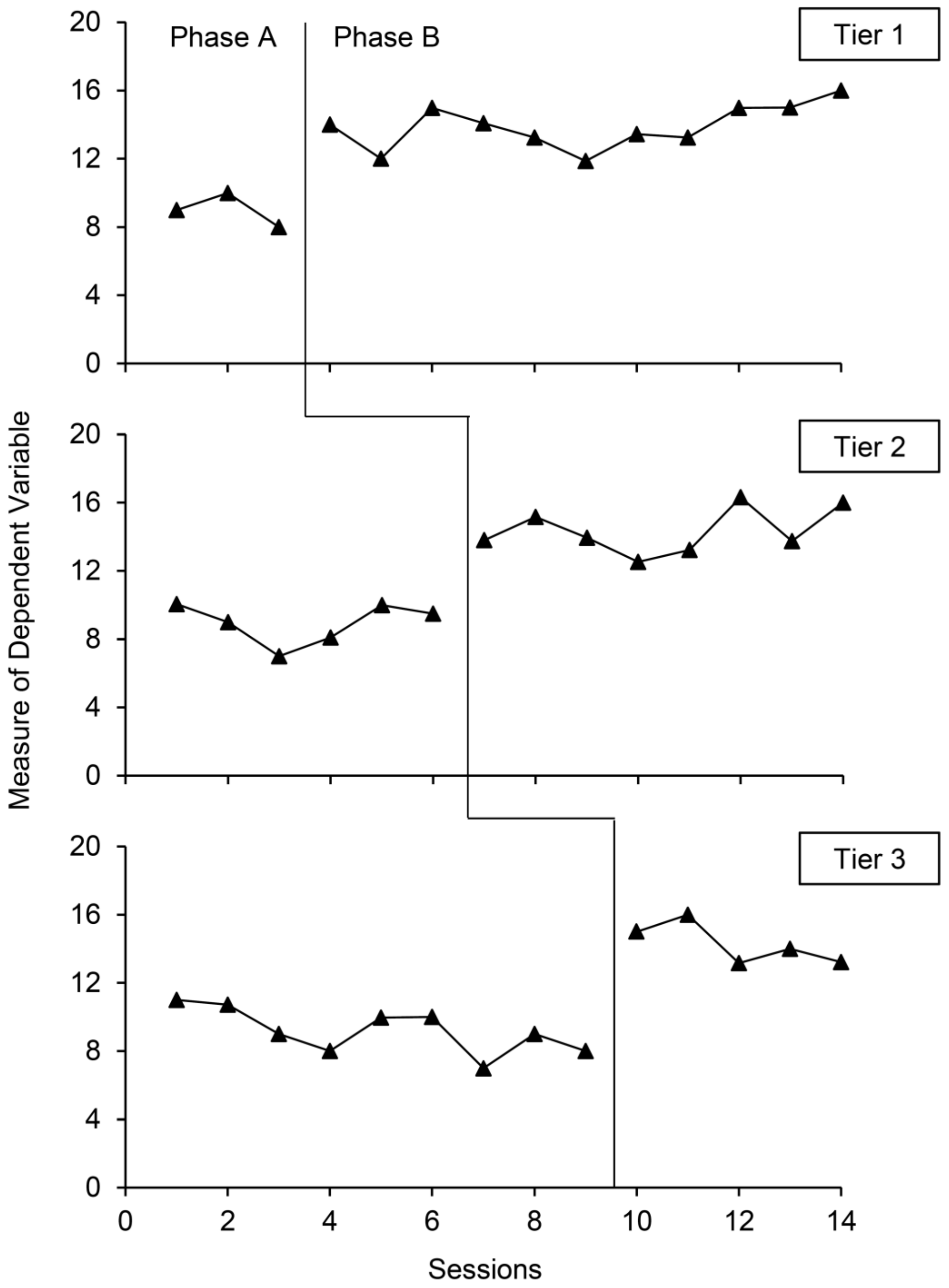
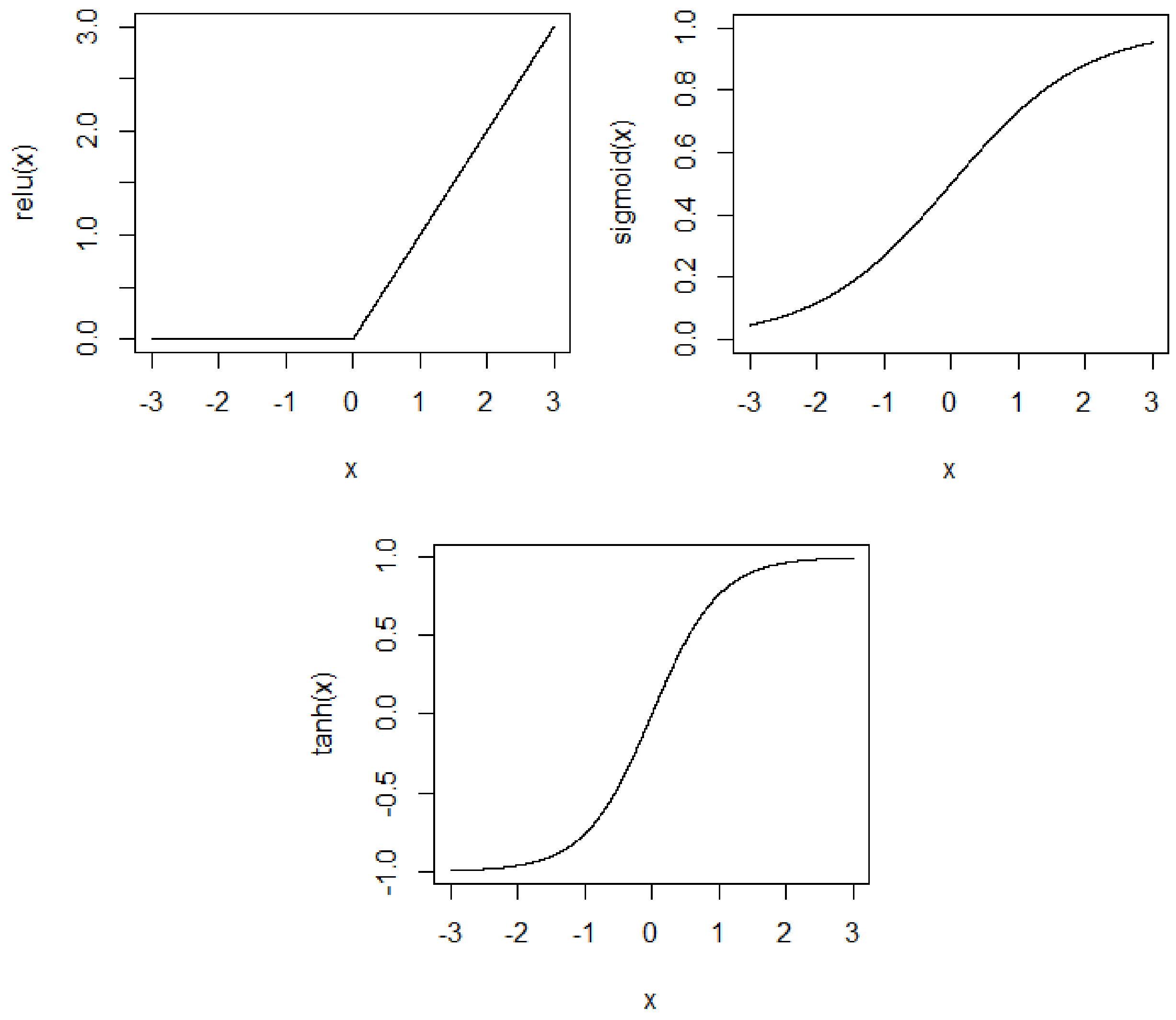


Figure 3

Graphs of Common Activation Functions for Binary Classification



Appendix A

Mathematical Foundations of Artificial Neural Networks

Example

Let’s assume that we want to train a dense artificial neural network with 24 features as input, 12 neurons in a single hidden layer, and one binary class label as output (as depicted in Figure 1).

In our example, a vector, **x**, represents our 24 features and a scalar, y, represents the binary class label. To facilitate comprehension, we provide a Python implementation of the equations below in our online repository (see the “ANN_from_scratch.py” file in https://osf.io/68kgy)

Step 1 – Initializing the Weight Matrices

Between layers, the neural network multiplies the values by weights. The initial step involves initializing the weight matrices. For our example, we need two weight matrices: one matrix between the input layer and the hidden layer, and another matrix between the hidden layer and the output layer. To reduce the number of values from 24 features (i.e., input) to 12 neurons in the hidden layer, the size of the first weight matrix, $\mathbf{W}_1$, is 24 rows by 12 columns. In contrast, the final layer produces an output of one class label from 12 neurons, which requires a second weight matrix, $\mathbf{W}_2$, of 12 rows by 1 column. The initialization involves randomly populating the rows and columns with values from a normal distribution with a mean of 0 and a standard deviation of 0.01.

Step 2 – Implementing Forward Propagation

During forward propagation, the input values move through each layer by being multiplied by weights and transformed by activation functions (see Figure 3 for examples of activation functions). The activation functions introduce nonlinearity by determining whether each neuron should fire or not (i.e., be included in the model or not). In the example, we will use a sigmoid function between both our layers.

Step 2.1. Computing the Hidden Activated Neurons

In this step, we first multiply the input data, **x**, with the weight matrix, $\mathbf{W}_1$, and add a bias vector, $\mathbf{b}_1$, which is initialized with zeros (see Equation 1). This bias vector functions like an intercept in a regression.

$$\mathbf{a_1} = \mathbf{xW_1} + \mathbf{b_1} \tag{1}$$

The result vector, $\mathbf{a}_1$, contains 12 neurons. The next step involves applying the sigmoid activation function to these neurons (see Equation 2).

$$\mathbf{h_1} = \frac{1}{1 + \mathrm{e}^{-\mathbf{a_1}}} \tag{2}$$

These computations lead to a vector, $\mathbf{h_1}$, that contains our activated neurons (12 activated neurons).

Step 2.2. Computing the Output

Based on our initial architecture, the neural network must produce a single output from 12 neurons. To do so, it multiplies the activated neurons from the hidden layer, $\mathbf{h_1}$, by the second weight matrix, $\mathbf{W}_2$, and adds a bias term (see Equation 3).

$$a_2 = \mathbf{h_1}\mathbf{W_2} + b_2 \tag{3}$$

The result is a scalar, a_2. The final step involves applying our activation function to produce the output (see Equation 4).

$$o = \frac{1}{1 + e^{-a_2}} \tag{4}$$

The output, o, is a value between 0 and 1.

Step 3 – Backpropagation of the Error

Step 3.1. – Applying Loss Function

The subsequent step involves computing the error, or loss, produced by the network and backpropagating its derivative to update the weights. A common loss function for binary outputs is the binary cross-entropy function (see Equation 5).

$$L = -(y\log(o) + (1 - y)\log(1 - o)) \tag{5}$$

The scalar, L, represents the loss of the network for our sample.

Step 3.2. – Updating the Output Weight Matrix (W_2)

However, we do not want to loss, but the differentiation of the loss to update the weights. We need to calculate the derivatives to backpropagate the gradient of the loss. The most straightforward way to differentiate matrices and vectors is to use the chain rule and begin by updating the last weight matrix, $\mathbf{W_2}$ (see Equation 6 for chain rule).

$$\frac{\partial L}{\partial \mathbf{W_2}} = \frac{\partial a_2}{\partial \mathbf{W}_2}\frac{\partial o}{\partial a_2}\frac{\partial L}{\partial o} \tag{6}$$

Now, we can compute each partial derivative separately (see Equations 7 to 9).

$$\frac{\partial \mathrm{a}_2}{\partial \mathbf{W}_2} = \frac{\partial}{\partial \mathbf{W}_2}[\mathbf{h_1 W}_{\mathbf{2}} + \mathrm{b}_2] = \mathbf{h}_{\mathbf{1}}^{\mathbf{T}} \tag{7}$$

$$\frac{\partial \mathrm{o}}{\partial \mathrm{a}_2} = \frac{\partial}{\partial \mathrm{a}_2}\left[\frac{1}{1+\mathrm{e}^{-\mathrm{a}_2}}\right] = \frac{\mathrm{e}^{-\mathrm{a}_2}}{(1+\mathrm{e}^{-\mathrm{a}_2})^2} = \frac{1}{1+\mathrm{e}^{-\mathrm{a}_2}} \cdot \left[1 - \frac{1}{1+\mathrm{e}^{-\mathrm{a}_2}}\right] = \mathrm{o}(1-\mathrm{o}) \tag{8}$$

$$\frac{\partial \mathrm{L}}{\partial \mathrm{o}} = \frac{\partial}{\partial \mathrm{o}}[-(\mathrm{y}\log(\mathrm{o}) + (1-\mathrm{y})\log(1-\mathrm{o}))] = \frac{\mathrm{o}-\mathrm{y}}{\mathrm{o}(1-\mathrm{o})} \tag{9}$$

If we put them in the same equation using our chain rule, it will produce the derivative of the loss function for the output weights (see Equation 10)

$$\frac{\partial \mathrm{L}}{\partial \mathbf{W_2}} = \mathbf{h}_{\mathbf{1}}^{\mathbf{T}} \cdot \mathrm{o}(1-\mathrm{o}) \cdot \frac{\mathrm{o}-\mathrm{y}}{\mathrm{o}(1-\mathrm{o})} = \mathbf{h}_{\mathbf{1}}^{\mathbf{T}}(\mathrm{o}-\mathrm{y}) \tag{10}$$

To update the weights, we multiply this derivative by the learning rate, α, and subtract the result from the original weights (see Equation 11).

$$\mathbf{W_2} = \mathbf{W_2} - \propto \mathbf{h}_{\mathbf{1}}^{\mathbf{T}}(\mathrm{o}-\mathrm{y}) \tag{11}$$

The learning rate indicates how fast the model learns, or updates itself, to fit the patterns.

If we repeat the same exercise with the bias term, b_2, we obtain Equation 12 to update it.

$$\mathrm{b}_2 = \mathrm{b}_2 - \propto (\mathrm{o}-\mathrm{y}) \tag{12}$$

Step 3.2. – Backpropagate to the First Weight Matrix (W_1)

Once again, we use the chain rule to facilitate the differentiation, but we apply it to the first weight matrix (see Equation 13).

$$\frac{\partial \mathrm{L}}{\partial \mathbf{W_1}} = \frac{\partial \mathbf{a_1}}{\partial \mathbf{W}_1}\frac{\partial \mathbf{h_1}}{\partial \mathbf{a_1}}\frac{\partial \mathrm{L}}{\partial \mathbf{h_1}} = \frac{\partial \mathbf{a_1}}{\partial \mathbf{W}_1}\frac{\partial \mathbf{h_1}}{\partial \mathbf{a_1}}\frac{\partial \mathrm{a}_2}{\partial \mathbf{h_1}}\frac{\partial L}{\partial \mathrm{a}_2} = \frac{\partial \mathbf{a_1}}{\partial \mathbf{W}_1}\frac{\partial \mathbf{h_1}}{\partial \mathbf{a_1}}\frac{\partial \mathrm{a}_2}{\partial \mathbf{h_1}}\frac{\partial o}{\partial \mathrm{a}_2}\frac{\partial \mathrm{L}}{\partial \mathrm{o}} \tag{13}$$

In Equation 13, the chain rule is applied in a way to facilitate the calculation of our derivatives. That is, we can readily compute the partial derivatives for the first three terms (see Equations 14 to 16) and already have the partial derivatives for the last two terms the from our prior computations (see Equations 8 and 9).

$$\frac{\partial \mathbf{a_1}}{\partial \mathbf{W}_1} = \frac{\partial}{\partial \mathbf{W}_1}[\mathbf{xW}_{\mathbf{1}} + \mathbf{b_1}] = \mathbf{x^T} \quad (14)$$

$$\frac{\partial \mathbf{h_1}}{\partial \mathbf{a_1}} = \frac{\partial}{\partial \mathbf{a_1}}\left[\frac{1}{1+\mathrm{e}^{-\mathbf{a}_1}}\right] = \frac{\mathrm{e}^{-\mathbf{a}_1}}{(1+\mathrm{e}^{-\mathbf{a}_1})^2} = \frac{1}{1+\mathrm{e}^{-\mathbf{a}_1}} \cdot \left[1 - \frac{1}{1+\mathrm{e}^{-\mathbf{a}_1}}\right] = \mathbf{h_1}(1-\mathbf{h_1}) \quad (15)$$

$$\frac{\partial \mathrm{a}_2}{\partial \mathbf{h_1}} = \frac{\partial}{\partial \mathbf{h_1}}[\mathbf{h_1 W}_{\mathbf{2}} + \mathbf{b_2}] = \mathbf{W_2^T} \quad (16)$$

To finalize our derivative, we combine all the terms together using the chain rule (see Equation 17)

$$\frac{\partial \mathrm{L}}{\partial \mathbf{W_1}} = \mathbf{x^T} \cdot \mathbf{h_1}(1-\mathbf{h_1}) \cdot \mathbf{W_2^T} \cdot (\mathrm{o}-\mathrm{y}) \quad (17)$$

Updating the weights will thus involve Equation 18.

$$\mathbf{W_1} = \mathbf{W}_{\mathbf{1}} - \propto \cdot\, \mathbf{x^T} \cdot \mathbf{h_1}(1-\mathbf{h_1}) \cdot \mathbf{W_2^T} \cdot (\mathrm{o}-\mathrm{y}) \quad (18)$$

As you can observe, this derivative contains the loss from the output layer as well its own loss, which is why it is referred to as backpropagation.

If we apply the same logic to the first bias vector, we can also update its values (see Equation 19).

$$\mathbf{b_1} = \mathbf{b}_{\mathbf{1}} - \propto \cdot\, \mathbf{h_1}(1-\mathbf{h_1}) \cdot \mathbf{W_2^T} \cdot (\mathrm{o}-\mathrm{y}) \quad (19)$$

Step 4 – Repeating the process

Step 4 involves repeating steps 2 and 3 (i.e., Equations 1 to 19) in a loop for a specific number of iterations. This number of iterations is called epochs. Running epochs will reduce the binary cross-entropy loss of the neural network.

To show how the computations operate, our example contained a single sample. However, multiple samples will influence the weights during typical training.

The primary reference for the development of this Appendix was *Data mining: Practical machine learning tools and techniques* (4th ed.) by I. H. Witten, E. Frank, M.A. Hall, & C. J. Pal (2017). For a more comprehensive review of the mathematical foundations of artificial neural networks, we recommend that you consult this source.

Appendix B

Installing Python and Packages

Step 1 – Installing a Python Distribution

Download and install a Python distribution. We strongly recommend the free version (individual edition) of Anaconda available at www.anaconda.com. The steps below assume that you are using Anaconda.

Step 2 – Creating a New Virtual Environment

You need to create an environment that will have all the packages. To create the environment, write the following code Terminal (Apple or Linux) or Anaconda Prompt (Windows):

```
conda create -n tfenv python=3.7
conda activate tfenv
```

From now on, you should write the following code when you open Terminal or Anaconda Prompt to ensure that you are working in the correct environment:

```
conda activate tfenv
```

The last line of your Anaconda Prompt or Terminal screen should begin with <tfenv>. If it begins with <base>, you have not activated your environment correctly.

Step 3 – Installing Packages

To install the necessary packages. Run the following code sequentially. Whenever you are prompted to Proceed, press on "y" followed by the enter key:

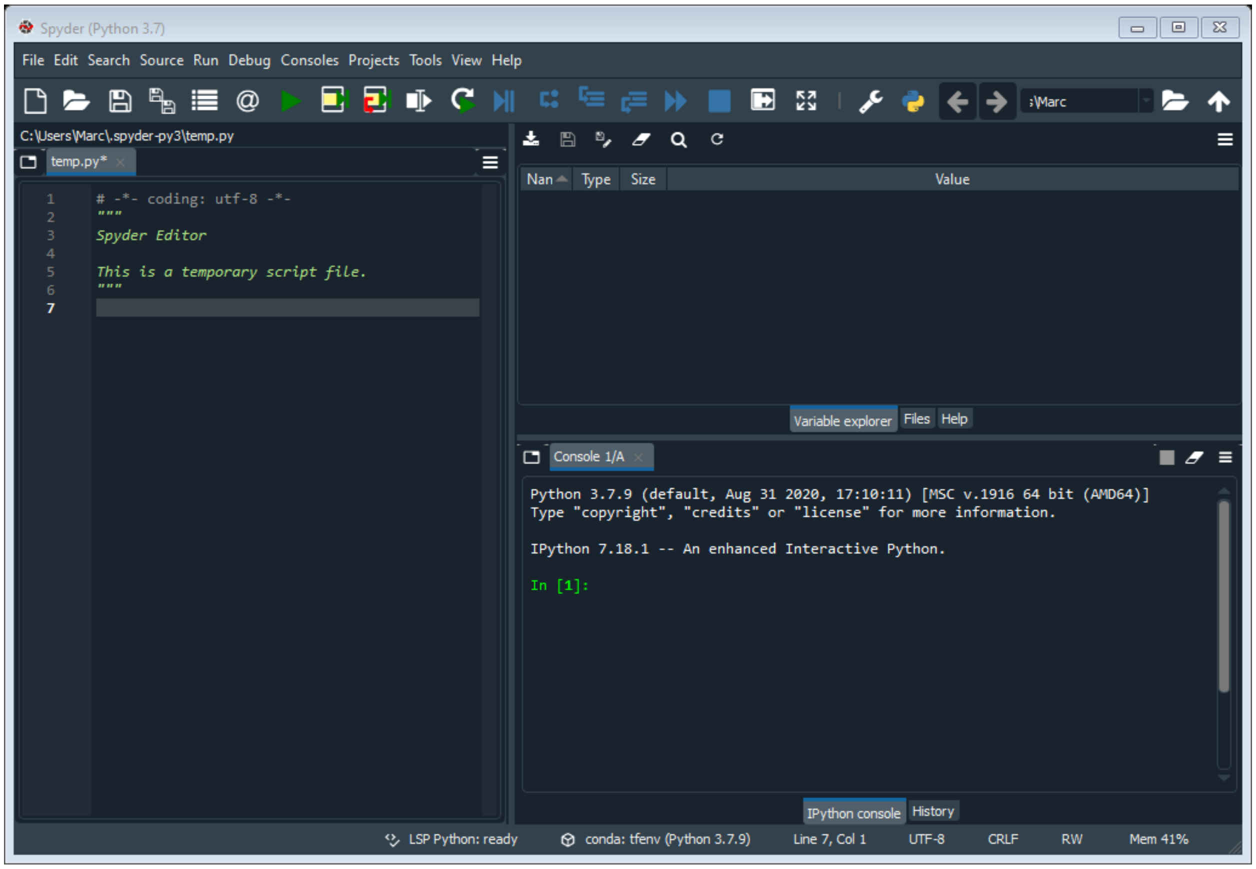
```
conda install spyder
conda install pandas
conda install scikit-learn
conda install tensorflow
```

Step 4 – Opening the Integrated Developmental Environment

To write and run your Python code, we suggest that you the Spyder integrated environment. To open spyder, write the following code

```
spyder
```

The following screen should appear:



```
# -*- coding: utf-8 -*-
"""
Spyder Editor

This is a temporary script file.
"""

```

```
Python 3.7.9 (default, Aug 31 2020, 17:10:11) [MSC v.1916 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.18.1 -- An enhanced Interactive Python.

In [1]:
```

You should first set the working the directory. The working directory is the folder in which you have downloaded or saved your data (.csv) files. To do so, you may click on the opened folder icon in the upper right corner and select the appropriate folder.

Enter all the Python code in the left panel of the screen. To run your code, highlight it and click on the “Run selection or current line” button in the toolbar ( ). When you run code, any warnings, errors, or printed information will appear in the console (lower right panel). You may also explore the variables in the Variable Explorer in the upper right panel.

Note that all code entered in Python is case sensitive.

This appendix was adapted with permission from “Tutorial: Applying machine learning in behavioral research” by S. Turgeon and M. J. Lanovaz, 2020, *Perspectives on Behavior Science* (https://doi.org/10.1007/s10803-020-04735-6). © Association for Behavior Analysis International.