Vignette DNNcausal

Mohammad Ghasempour

May 2024

Contents

1 Introduction

DNNcausal is a package that performs doubly robust estimation of the average treatment effect (ATE) and the average treatment effect on the treated (ATT) by fitting neural networks to the nuisance functions. For more theoretical details see Ghasempour et al. (2024). The package is written based on Keras (Chollet et al., 2015) in R (R Core Team, 2021), which is a high-level API to build and train a vast range of deep neural network models. Keras is modular and simple.

Creating Keras models can be done by connecting configurable building blocks. Also, it is easy to create custom building blocks for new ideas.

1.1 Installation

To install and load the DNNcausal package in R from GitHub, run the following commands:

```r
install.packages("devtools")
library(devtools)
install_github("stat4reg/DNNcausal")
```

Then you can load the library in the following way, and the first time you need also to install Keras as well.

```r
library(DNNcausal)
keras::install_keras()
```

The last line will install a version of Python and the Python package Keras on the machine. This is a necessary step, since the R code written with the commands from the Keras R package will be translated to Python code and run on the installed version. If there is an existing version of Python and the Keras Python package on your machine, instead of the last two lines you can specify the path to the current Python first and then load the DNNcausal package.

```r
library(reticulate)
use_python("/path/to/python")
library(DNNcausal)
```

2 Estimator of ATE and ATT

2.1 Treatment effect

A central problem in the field of causal inference is to identify and estimate the effect of treatment T on a given outcome of interest Y in an observation study, where there is the risk of having confounders X that affect both treatment and the outcome of interest. Rubin's framework (Rubin, 1974, 1991) is used to tackle this problem. In this framework, for a binary treatment, we consider different potential outcomes, Y_0 and Y_1. These variables are the outcome of interest in the presence of treatment exposures $T = 0$, and $T = 1$, where we assume $Y = TY_1 + (1 - T)Y_0$. The effect can therefore be determined by comparing Y_0 and Y_1. However, it is not possible that both of the variables are directly observed for one individual. This issue is recognized as the Fundamental Problem of Causal Inference. Therefore we can target the average of contrasts instead i.e. the Average Treatment Effect (ATE): $E(Y_1 - Y_0) = E(Y_1) - E(Y_0)$. Furthermore, we might be interested in the average for only a certain segment of the population, such as those who are exposed to the treatment $T = 1$. In this case, the parameter of interest is the Average Treatment Effect on Treated (ATT): $E(Y_1 - Y_0|T = 1) = E(Y_1|T = 1) - E(Y_0|T = 1)$. If we could do a randomized study, we would be able to assign the treatment levels randomly and then the average of outcomes in each of the treatment groups would be unbiased estimations of $E(Y_1)$ and $E(Y_0)$. Observational studies, on the other hand, involve predefined treatment levels, and because the treatment assignment mechanism is not independent of the potential outcomes, sample averages will be biased. Assuming strong ignorability, i.e. observing all confounders is the solution in this case. See the Rosenbaum and Rubin (1983); Rubin (1980) for more details on the assumptions. The total law of expectations lets us represent the parameter of interest with the average of expectations conditional on the set of potential confounders: $E(Y_t) = E(E(Y|X, T = t))$, for $t = 0, 1$. Having data-driven fits of the regressions (conditional expectations) at hand enables the use of a plug-in estimator of the causal parameters. The performance and bias of these estimators depends on the choice of the regression model. Therefore, we use, in this package, more advanced estimators to achieve better performance while keeping requirements low.

2.2 AIPW estimator

The estimators we have implemented in the package are known as Augmented inverse probability weighted (AIPW) and are made by modification of a plug-in estimator to ensure asymptotically unbiasedness. They have been thoroughly studied for both problems of ATE and ATT (Farrell et al., 2021; Robins et al., 1994). AIPW estimators have a so called doubly robustness property. This ensures that asymptotic results we know for the estimator hold whenever our estimations of nuisance parameters converge at a $n^{-1/4}$ rate to the true nuisance functions, where n is the sample size. In addition, the nuisance estimators can also compensate for each other's rates. For the parameter ATE, we need to have a fitted model for these three nuisance functions: the potential outcome models: $m_0(X) := E(Y|X, T = 0)$, $m_1(X) := E(Y|X, T = 1)$, and the propensity score model: $p(X) := E(T = 1|X)$. When the parameter of interest is ATT, only two nuisance models are needed, the propensity score and one of the outcome models. If we know the correctly specified parametric model for one of these nuisance parameters, then we can use any misspecified for the other one as long as it does not violate the consistency of the estimator. Another possibility is to use machine learning models for both of them, then we have a slower rate for both of them. In general, when the product of their rates is as fast as root n, we expect the asymptotic results to hold.

In this package, with a random sample $X_1, \ldots, X_n$, we fit any kind of neural network specified by the user, to the nuisance functions to obtain $\hat{m}_0$, $\hat{m}_1$, $\hat{p}$. Then the AIPW estimator for these two parameters of interest are:

$$\widehat{\text{ATE}}_{\text{AIPW}} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}[T_i = 1]\frac{Y_i - \hat{m}_0(X_i)}{\hat{p}(X_i)} - \mathbf{1}[T_i = 0]\frac{Y_i - \hat{m}_1(X_i)}{1 - \hat{p}(X_i)},$$

and

$$\widehat{\text{ATT}}_{\text{AIPW}} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}[T_i = 1]\frac{Y_i - \hat{m}_0(X_i)}{n_1/n} - \mathbf{1}[T_i = 0]\frac{\hat{p}(X_i)}{1-\hat{p}(X_i)}\frac{Y_i - \hat{m}_0(X_i)}{n_1/n},$$

where n_1 is the size of the treated group with $T = 1$.

2.3 Estimation of the variation

To estimate the variation of the AIPW estimators, we use an estimator of the asymptotic variance. As discussed above, as long as we use flexible enough machine learning models to fit the potential outcome models and propensity score model, we expect the asymptotic variance to be close enough to the variation of our estimator for big enough sample sizes. To estimate the asymptotic variance we can use the sample variance of the influence function of our parameter of interest. The estimators are expressed here:

$$\begin{aligned}\widehat{\text{VAR}}_{\text{ATE}} = &- \widehat{\text{ATE}}^2_{\text{AIPW}} \\ &+ \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}[T_i = 1]\left[\frac{Y_i - \hat{m}_0(X_i)}{\hat{p}(X_i)}\right]^2 - \mathbf{1}[T_i = 0]\left[\frac{Y_i - \hat{m}_1(X_i)}{1-\hat{p}(X_i)}\right]^2\end{aligned}$$

$$\begin{aligned}\widehat{\text{VAR}}_{\text{ATT}} = -\widehat{\text{ATT}}^2_{\text{AIPW}} + \frac{1}{n}\sum_{i=1}^{n}\Bigg[&-\bar{\hat{m}}_0 + \mathbf{1}[T_i = 1]\frac{Y_i - \hat{m}_0(X_i) + \bar{\hat{m}}_0}{n_1/n} \\ &- \mathbf{1}[T_i = 0]\frac{\hat{p}(X_i)}{1-\hat{p}(X_i)}\frac{Y_i - \hat{m}_0(X_i)}{n_1/n}\Bigg]^2,\end{aligned}$$

where $\bar{\hat{m}}_0 = \frac{1}{n}\sum_{i=1}^{n}\hat{m}_0(X_i)$.

3 Fitting neural models for nuisance functions

Deep learning neural network models approximate a function based on observed input and output. For example for a function f, where $y = f(x)$, a deep learning neural network defines a map $g(x, \omega)$, and learns ω in a way to approximate f as good as possible on a given set $\{(x_i, y_i); i = 1\ldots, n\}$. These models can be divided into two subcategories; Feedforward and Recurrent Neural Networks. A feedforward network resembles a directed acyclic graph in which the directions imply that the information flow begins at the input, passes through the intermediate neurons, and ultimately finishes at the output. The acyclic nature of these networks distinguishes them from the recurrence networks. There are feedback connections in recurrent networks, which means that information coming from a neuron can be transmitted back to the same neuron. Three different types of neural networks are discussed in this article, where the first two are feedforward networks, and the third is a recurrent network.

A feedforward network can be viewed as a chain of functions, where the output from one is the input to the next. In this case, the first function is referred to as the first layer, and the last one is referred to as the output layer. All other functions between these two layers are called hidden layers. Each of these functions is assumed to consist of a number of simple calculation units called neurons. Essentially, these units are functions that take a vector and return a scaler. We can first consider linear functions for these units, which then collapse into linear regression. In order to obtain a more capable model, we must also incorporate nonlinear functions in each neuron. These nonlinear components of the neuron are called activation functions, and they are often chosen in the same way throughout the entire network. More information about architectures and activation functions, as well as tips on how to select them, can be found in Goodfellow et al. (2016).

The DNNcausal package utilizes the Keras package in R, which allows the user to define their own neural network architecture. The simplest way of creating a neural network architecture in Keras is to use sequential models. Sequential model in Keras can be used in situations where an architecture is a stack of layers. For example, if a model is a fully connected network with one layer and 16 neurons for a regression problem, you can create your model in the following pipeline:

```r
model <- keras_model_sequential()

model %>%

  # Adds a densely-connected layer with 16 units to the model:
  layer_dense(units = 16, activation = "relu") %>%

  # Add the output layer:
  layer_dense(units = 1, activation = "relu")
```

The layers considered in this example are dense i.e. all neurons in one layer are connected to all neurons in the previous layer. There are many other layer types available in Keras to choose from. Depending on the type there are different arguments that the user needs to specify for building the layers. Although there are some arguments in common for all of the layer types. The most important argument is `activation`, the activation function. It has several options to choose from, such as `"relu"`, `"linear"`, `"sigmoid"`, `"softmax"`, and `"tanh"`. In case the user does not specify it, the default activation is `"linear"`.

Other common arguments in most layers relate to regularization. There are three arguments: `kernel_regularizer`, `bias_regularizer`, and `activity_regularizer`. The first and second penalize the layer weights, whereas the third penalizes the layer output. There are three regularization functions available in Keras that the user can choose from to assign to the above arguments. These are: `regularizer_l1`, `regularizer_l2`, and `regularizer_l1_l2`, which correspond to L1, L2, and elastic regularization. An example of L1 regularization for the layer kernel is as follows:

```r
  layer_dense(units = 16, activation = "relu", kernel_regularizer =
     regularizer_l1(0.01))
```

More specific arguments will be discussed separately in the following sections.

3.1 Neural network architecture

3.1.1 Fully connected neural networks

Creating a fully connected model can be done by stacking dense layers. More information about fully connected neural networks can be found in Hastie et al. (2001, Chapter 11). For example, in Figure 1 input and output are connected through two dense layers.

```
model_m <- keras_model_sequential()  %>%
    layer_dense(units = 128, activation = "relu", input_shape = 10)
     %>%
    layer_dense(units = 80, activation = "relu")

model_p <- keras_model_sequential() %>%
    layer_dense(units = 32, activation = "relu", input_shape = 10 )
     %>%
    layer_dense(units = 8, activation = "relu")
```

We define two sequential models here, one for the outcome model and one for the propensity score. When the potential outcome is a continuous variable, the package stacks the output layer with linear activation for a given model. The output layer of the propensity score model will have two neurons with softmax activation. The output layers will be added by the package, so users are required to define flat layers at the end of their models.

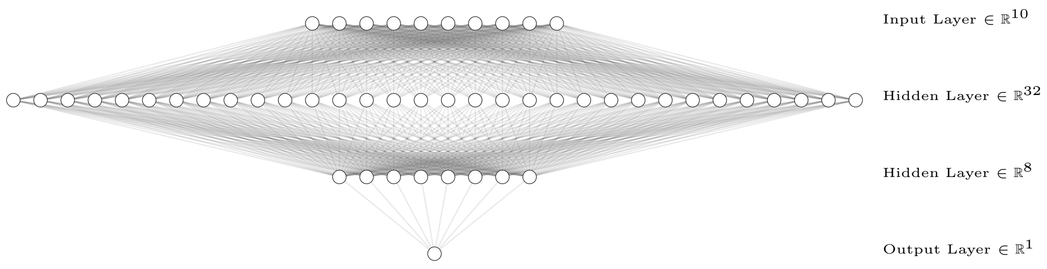

Input Layer $\in \mathbb{R}^{10}$
Hidden Layer $\in \mathbb{R}^{32}$
Hidden Layer $\in \mathbb{R}^{8}$
Output Layer $\in \mathbb{R}^{1}$

Figure 1: Fully connected neural network with 32 and 8 neurons in two hidden layers

3.1.2 Convolutional neural networks

Several types of convolutional neural networks (CNN) (Lecun et al., 1995) are available in Keras, including one, two, and three-dimensional. For example, time series are one-dimensional, while images are two-dimensional. Figure 2 visualizes an instance of a one-dimensional CNN, and the following code illustrates how this can be defined.

```
model_m <- keras_model_sequential() %>%
    layer_conv_1d(filters = 128, kernel_size = 4, padding = "valid"
    , activation = "relu", input_shape = c(k,1)) %>%
    layer_conv_1d(filters = 16, kernel_size = 3, padding = "same",
    activation = "relu") %>%
    layer_flatten()

model_p <- keras_model_sequential() %>%
    layer_conv_1d(filters = 32, kernel_size =4, padding = "valid",
    activation = "relu", input_shape = c(k,1 )) %>%
    layer_conv_1d(filters = 8, kernel_size =3, padding = "same",
    activation = "relu") %>%
    layer_flatten()
```

Convolutional layers have special arguments that need to be specified. The `filters` determines how many different filters will be applied to the input and how many output vectors it will produce. The second argument that needs to be set is the `kernel_size`, which determines the size of the kernel for all filters in that layer. For a one-dimensional convolution layer, the kernel size is a number representing the length of the kernel vector. However, for two-dimensional convolutions, it is a vector of size two, specifying the dimensions of the kernel matrix. The next argument is `strides`. If it is set to one, the kernel slides over the input one step at a time. If you want the kernel to move by more steps, you must set the `strides` argument to corresponding step size. The last argument discussed here is `padding`. Two possible values can be set: `"valid"` and `"same"`. Applying the kernels to the inputs typically produces output vectors that are shorter than the input. Using the `"same"` option ensures the output has the same length as the input by padding zeros on both sides of the input vector. The `"valid"` option means no padding is applied to the input vector.

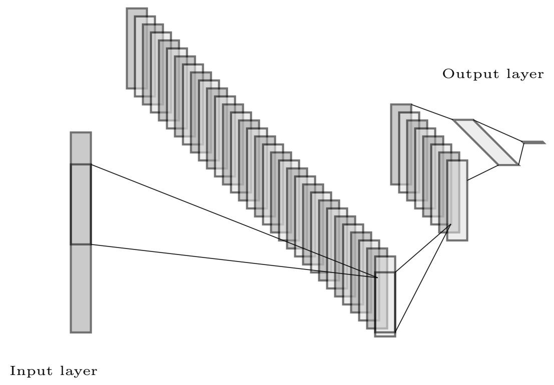


Figure 2: Convolutional neural network with 32 and 8 filters in two hidden layers

3.1.3 Recurrent neural networks

In this section, a recurrent neural network (RNN) is examplified. The most famous type is the Long Short-Term Memory (LSTM) network, which is used here. Earlier, it was mentioned how L1 or L2 regularization can be applied to each layer. In this example, another well-known type of regularization is utilized: dropout (Hinton et al., 2012). In order to apply dropout on neurons belonging to a layer in the sequential model, the `layer_dropout` function must be invoked immediately following that layer. An essential argument for this regularization is the `rate`, which is a value between 0 and 1 that determines the proportion of neurons to drop.

```
model_m <- keras_model_sequential() %>%
  layer_lstm(units = 50, return_sequences = TRUE, input_shape = c
    (1, 1)) %>%
  layer_dropout(rate = 0.2) %>%
  layer_lstm(units = 50, return_sequences = TRUE) %>%
  layer_dropout(rate = 0.2) %>%
  layer_lstm(units = 50) %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 1)
```

3.2 Loss

Fitting neural networks is achieved through gradient-based methods to minimize a loss function. This minimization occurs with respect to the weights of the neural network. The loss function measures, for a given set of weights, how far the predictions are from the true labels (response/outcome) in the dataset. Quantifying this distance can be done in various ways and is highly dependent on the problem and the type of data we have. For classification problems, the usual choice is binary cross-entropy, which can also be used for propensity scores. In regression problems, the usual choice is the mean square error, which can also be used for continuous outcome models. For the ATE case, however, we implemented a causal loss as described in Farrell et al. (2021).

3.3 Optimizer

As mentioned above, an optimization method is needed for solving the minimization problem. Keras allows for a wide range of optimizers that the user can choose from. Here is the list with the default values of the parameters:

```
SGD(lr = 0.01, momentum = 0, decay = 0, nesterov = FALSE,
  clipnorm = -1, clipvalue = -1)
RMSprop(lr = 0.001, rho = 0.9, epsilon = 1e-08, decay = 0,
  clipnorm = -1, clipvalue = -1)

Adagrad(lr = 0.01, epsilon = 1e-08, decay = 0, clipnorm = -1,
  clipvalue = -1)

Adadelta(lr = 1, rho = 0.95, epsilon = 1e-08, decay = 0,
  clipnorm = -1, clipvalue = -1)
```

```

Adam(lr = 0.001, beta_1 = 0.9, beta_2 = 0.999, epsilon = 1e-08,
  decay = 0, clipnorm = -1, clipvalue = -1)

Adamax(lr = 0.002, beta_1 = 0.9, beta_2 = 0.999, epsilon = 1e-08,
  decay = 0, clipnorm = -1, clipvalue = -1)

Nadam(lr = 0.002, beta_1 = 0.9, beta_2 = 0.999, epsilon = 1e-08,
  schedule_decay = 0.004, clipnorm = -1, clipvalue = -1)
```

3.4 Other hyperparameters

`Epochs` and `batch_size` are hyperparameters that can be set for fitting neural network models. The number of epochs refers to how many times, on average, each data point is used for gradient reduction. The good performance of all gradient-based optimization methods listed above owes to their stochastic property. This means that, at each step of gradient reduction, instead of using all the data, a random sample is used to form the loss function and calculate the gradient step reduction. The size of these samples is called the batch size.

4 Example and code

4.1 Functions

The package DNNcausal contains two main functions for estimating the ATE and the ATT:

```
  aipw.dnn(Y,T,X_t,X,rescale_outcome,do_standardize,use_scalers,
    rescale_treated,model, optimizer,loss,epochs,batch_size,
    propensity_score,debugging,verbose)
  aipw.att(Y,T,X_t,X,rescale_outcome,do_standardize,use_scalers,
    rescale_treated,model, optimizer,loss,epochs,batch_size,
    propensity_score,debugging,verbose)
```

Both functions use neural networks to perform estimation of the nuisance models and AIPW estimation of the parameter of interest. The input arguments can be categorized into three different groups. The first contains the raw data: covariates, treatment status, observed outcomes, and data manipulation options. The second set of arguments includes neural network models and corresponding hyperparameters. The third group consists of miscellaneous parameters which we explain further.

4.1.1 Data inputs

There are four variables in the raw data set: `Y`, `T`, `X_t`, and `X`.

- `Y`: is a numerical vector of observed outcomes of length n.
- `T`: is a logical vector of treatment statuses of length n.

- X_t: is the set of covariates. If covariates are time series, it should be a list of k different n * p matrixes. Here p is the length of the time series, and k is the number of different covariates. If covariates are single valued, it should be a matrix of size n * k.

- X: if there are some single value covariates besides the time series, it can be considered here. It has to be k * n matrix. The default value is NULL.

- rescale_outcome: determine if scale the outcome values to have zero mean and one standard deviation. Default is True.

- do_standardize: determine if standardize the covariates row-wise or column-wise or not at all. Options are "column" and "row". Default is NULL.

- use_scalers: in the case that covariates are times series, determine if to use the mean and SD of time series after the standardization and other single value covariates in an additional parallel neural network or not. Default is True.

- rescale_treated: In the case that scalers are used, determine if the scaling considers all covariates or just treated ones.

4.1.2 Neural Network inputs

The following inputs are needed to make the neural network models for each of the nuisance models.

- model: It can be defined by a vector of size 3 (for aipw.dnn function) or size 2 (for aipw.att function), or it can be just one model. In the latter case, that one model will be used for all nuisance models. The default value for model is NULL, and in this case, the functions will use a specific predefined network. Each model should be defined using the package Keras. One example of a model defined using Keras would be:

```
model <- keras_model_sequential() %>%
        layer_dense(units = 30,activation = "relu",input_shape
    = 10) %>%
        layer_dense(units = 20,activation = "relu")
```

  The above example is a fully connected neural network with two hidden layers. The network need a vector of dimension ten as input and it contains 30 and 20 neurons in the hidden layers in the first and second hidden layers, respectively. An example of a CNN model is:

```
model <- c(keras_model_sequential() %>%
layer_conv_1d(128,5,padding = "same",activation = "relu",input
    _shape = c(10,7))%>%
        layer_conv_1d(model_m,256,3,padding = "valid",
    activation = "relu")
, keras_model_sequential() %>%
layer_conv_1d(32,5,padding = "same",activation = "relu",input_
    shape = c(10,7))%>%
        layer_conv_1d(model_m,64,3,padding = "valid",
    activation = "relu"))
```

It is not needed to define the output layer for the model. The package defines the last layer properly for each of the nuisance models.

- `Optimizer`: can be a vector containing two optimizers or just one. In the presence of two optimizers, the first one will be used for the outcome models, and the second one will be used for the propensity score estimation. The default is `NULL`, and in this case, Adam optimizer is used. Example:

```
    optimizer <- c(optimizer_adam(lr = .003), optimizer_adam(
    lr = .003))
```

- `Loss`: can be a vector containing two loss functions or just one. In the presence of two functions, the first one will be used for the outcome models, and the second one will be used for the propensity score estimation. The default is `NULL`, and in this case, mean square error and cross-entropy error are used for the outcome models and propensity score, respectively. Example:

```
    loss <- c(loss_mean_squared_error(), loss_binary_
    crossentropy())
```

- `Epochs`: default is 256. A vector is also acceptable.
- `batch_size`: defaults is 200. A vector is also acceptable.

4.1.3 Miscellaneous inputs

- `propensity_score` can be used to specify pre-estimated propensity scores. When set, the functions will not fit the neural network for estimating the propensity score, and the predefined values will be used in the final estimation.
- `debugging` is a logical variable, and if it is `TRUE`, the function will return the estimated vectors for the outcome models and the propensity score.
- `verbose` can be a vector of logical variables or just one. That will control the verbosity of the fitting process.

4.2 Examples

We consider the study in the Ghasempour et al. (2024), where the following Data-Generating Process (DGP) is defined. The relevant code to generate a sample is this R function:

```r
DGP <- function(N) {

  # generate the covariates
  x1 <- rnorm(N, 100, 20)
  x2 <- rnorm(N, 102, 15)
  x3 <- rnorm(N, 105, 13)
  x4 <- rnorm(N, 107, 11)
  x5 <- rnorm(N, 109, 8)
  x6 <- rnorm(N, 110, 20)
  x7 <- rnorm(N, 112, 15)
  x8 <- rnorm(N, 115, 13)
  x9 <- rnorm(N, 117, 11)
  x10 <- rnorm(N, 119, 8)

  # define the non-linear step functions that will be used in the outcome models
  I1 <- function(x,y,z,w){
    return(10*(((y-x)/x) > .15 & ((z-y)/y) > .15 & ((w-z)/z) > .15))
  }
  I2 <- function(x,y,z,w){
    return(5*(((y-x)/x) < .05 & ((z-y)/y) < .05 & ((w-z)/z) < .05))
  }
  I3 <- function(x,y,z,w){
    return(3*(sign(y - 1.1*x)*sign(z - 1.1*y) ==-1 & sign(y - 1.1*x)*sign(z - 1.1*y) ==-1) )
  }

  # define the outcome models
  f0 <- function(x1,x2,x3,x4,x5,x6,x7,x8,x9,x10){
    return(1 + 1*(I1(x1,x2,x3,x4) + I2(x4,x5,x6,x7) + I3(x6,x7,x8,x9) ))
  }
  f1 <- function(x1,x2,x3,x4,x5,x6,x7,x8,x9,x10){
    return(2 - 1*(I1(x1,x2,x3,x4) + I2(x4,x5,x6,x7) +I3(x6,x7,x8,x9) ))
  }

  # call the outcome models on generated covariates
  m0 <- f0(x1,x2,x3,x4,x5,x6,x7,x8,x9,x10)
  m1 <- f1(x1,x2,x3,x4,x5,x6,x7,x8,x9,x10)

  # generate outcome errors from random variables with a normal distribution
  e0 <- rnorm(N, mean = 0, sd = 1)
  e1 <- rnorm(N, mean = 0, sd = 1)

  # simulate the potential outcomes
  y0 <- m0 + e0
  y1 <- m1 + e1

  # generate treatment probabilities
  prob <- 1/(1+1*exp(.1*(0.05*x5 - I1(x1,x2,x3,x4) + I2(x4,x5,x6,x7) +  I3(x6,x7,x8,x9)    )  ))
  # simulate the exposure level based on the probabilities
  Tr <- rbinom(N, 1, prob)

  # form the observed outcome
  Y <- Tr * y1 + (1 - Tr) * y0

  # return the list of covariates, observed outcome, and treatment
    level
  return(list(x=cbind(x1=x1,x2=x2,x3=x3,x4=x4,x5=x5,x6=x6,x7=x7,x8=
    x8,x9=x9,x10=x10),y0=y0,y1=y1, T=Tr, p = prob,y=Y))
}
```

Now we can define different models for estimating ATT under the above data-generating process. For the fully connected network, we can use the `model_p` and `model_m` that have been defined in the code chunk in Section 3.1.1. We first write a function that gives us these models:

```r

FullyConnectedModel <- function(){

    # define the models using keras sequential model
    model_m <- keras::keras_model_sequential() %>%
        keras::layer_dense(units = 128, activation = "relu", input_
    shape = 10) %>%
        keras::layer_dense(units = 80, activation = "relu")

    model_p <- keras::keras_model_sequential() %>%
        keras::layer_dense(units = 32, activation = "relu",input_
    shape = 10) %>%
        keras::layer_dense(units = 8, activation = "relu")

    return(c(model_m,model_p))
}
```

We compose similar functions for the other two neural network methods as well. The functions create two neural models for the nuisance parameters.

```r

ConvolutionalModel <- function(){

    # define the models using keras sequential model
    model_m <- keras::keras_model_sequential() %>%
        keras::layer_conv_1d(filters = 128, kernel_size = 4,
    padding = 'valid', activation = "relu", input_shape = c(10,1))
    %>%
        keras::layer_conv_1d(filters = 16, kernel_size = 3, padding
     = 'same', activation = "relu") %>%
        keras::layer_flatten()

    model_p <- keras::keras_model_sequential() %>%
        keras::layer_conv_1d(filters = 32, kernel_size = 4, padding
     = 'valid', activation = "relu", input_shape = c(10,1 )) %>%
        keras::layer_conv_1d(filters = 8, kernel_size = 3, padding
    = 'same', activation = "relu") %>%
        keras::layer_flatten()

    return(c(model_m,model_p))
}
```

```r

```

```r
LSTMModel <- function(){

    # define the models using Keras sequential model
    model_m <- keras_model_sequential() %>%
        layer_lstm(units = 128, return_sequences = TRUE, input_shape = c(10, 1)) %>%
        layer_dropout(rate = 0.2) %>%
        layer_lstm(units = 32, return_sequences = TRUE) %>%
        layer_dropout(rate = 0.2) %>%
        layer_flatten()

    model_p <- keras_model_sequential() %>%
        layer_lstm(units = 32, return_sequences = TRUE, input_shape = c(10, 1)) %>%
        layer_dropout(rate = 0.2) %>%
        layer_lstm(units = 8, return_sequences = TRUE) %>%
        layer_dropout(rate = 0.2) %>%
        layer_flatten()

    return(c(model_m,model_p))
}
```

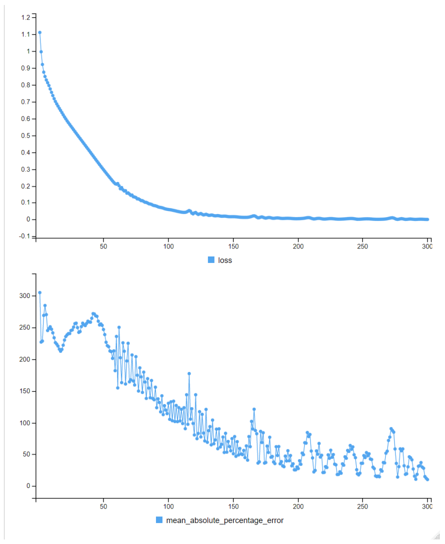
Lastly, we need to create a function that specifies the optimization method to be able to fit our neural network models.

```r
optimizer <- function(){
    optimizer_m <- keras::optimizer_adam(lr = 0.003)
    optimizer_p <- keras::optimizer_adam(lr = 0.003)
    return(c(optimizer_m, optimizer_p))
}
```

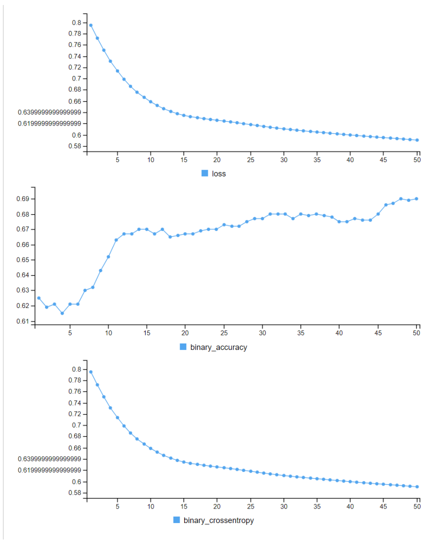
Using each of these neural network models, we perform an estimation of ATT with the following scripts. The first estimation is called `aipwfNN`. The function `aipw.att` is used for this estimation, when the fully connected neural network model is generated using the `FullyConnectedModel` function we created earlier.

```r

    n <- 1000
    k <- 10
    seed <- 1169876
    set.seed(seed)

    # generate data with the DGP function
    data = DGP(n)

    # perform the FNN estimator
    aipwfNN <- aipw.att(data$y, as.logical(data$T), X_t = data$x, model =FullyConnectedModel(), optimizer = optimizer(), epochs = c(100,50), batch_size = 1000, verbose = FALSE, use_scalers = FALSE, debugging = FALSE, rescale_outcome = TRUE, do_standardize = "Column")
```

When the verbose parameter is set to `TRUE`, this function generates Figure 3 and returns the corresponding values in the console for each iteration of the learning process (epoch). Upon fitting the outcome model, it prints the loss value at the end of each epoch, as well as the mean absolute percentage error, which is equal to: $\frac{1}{n}\sum_{i=1}^{n}|\frac{Y_i-\hat{Y}_i}{Y_i}|$. This is illustrated in Figure 3a. These Figures



(a) The outcome model training



(b) The propensity score training

Figure 3: Errors and accuracy of fits during the learning processes of the nuisance functions using fully connected neural networks

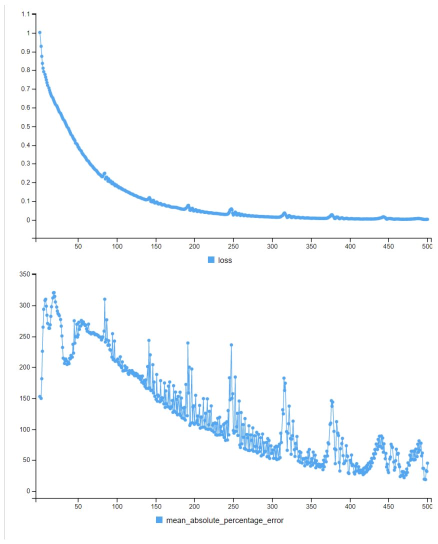
can help if it is needed to decide on the number of epochs. As we can see in this example, the loss function does not decrease significantly after the 100th epoch. However, as there is a decreasing trend in percentage error, we can decide to keep these 500 epochs. When fitting the propensity score, it returns the loss value, binary accuracy, and binary cross-entropy. The binary accuracy is the percentage of matched labels among all samples. (See Figure 3b)

The second architecture we consider is a convolutional neural network, using the function `ConvolutionalModel`. See Figure 4 for the learning process plots. A comparison of the binary accuracy plot of the propensity scores for convolutional and fully connected shows that the convolutional model reaches the same level of accuracy much faster than the fully connected when it comes to this data-generating process and binary cross-entropy loss.

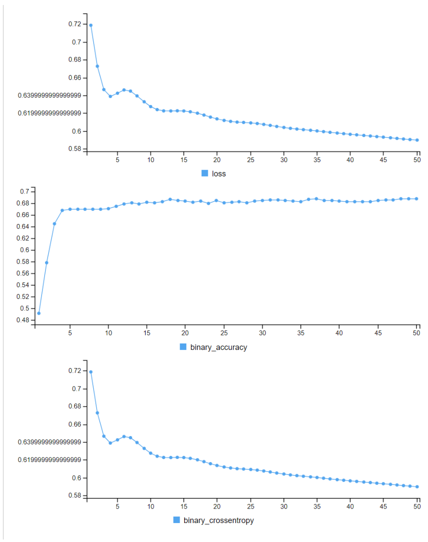
```
  # perform the CNN estimator
  aipwCNN <- aipw.att(data$y, as.logical(data$T), X_t = list(data
  $x), model = ConvolutionalModel(), optimizer = optimizer(),
  epochs = c(100,50), batch_size = 1000, verbose = FALSE, use_
  scalers = FALSE, debugging = FALSE, rescale_outcome = TRUE, do_
  standardize ="Column" )
```

Finally, we use a recurrent neural network; the long short-term memory (LSTM) model. The function `LSTMModel` was created to do this. In Figure 5 learning graphs for this case are depicted.

```
  # perform the LSTM estimator
  aipwLSTM <- aipw.att(data$y, as.logical(data$T), X_t = data$x,
  model = LSTMModel(), optimizer = optimizer(), epochs = c
```



(a) The outcome model training



(b) The propensity score training

Figure 4: Errors and accuracy of fits during the learning processes of the nuisance functions using convolutional neural networks

```
    (100,50), batch_size = 1000, verbose = FALSE, use_scalers =
    FALSE, debugging = FALSE, rescale_outcome = TRUE, do_
    standardize = "Column")
```

In accordance with the data-generation process, the treatment effect is constant -3 for all individuals, therefore, ATE and ATT are -3 as well. We present in Table 1 the estimates and the variation estimates for the $\widehat{\text{AIPW}}_{\text{ATT}}$ using each of the architectures.

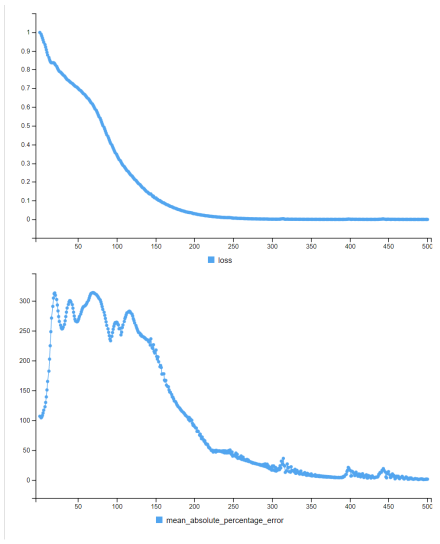
Table 1: Estimation of ATT, and the standard deviation using different neural network architectures to fit nuisance functions

	Est	Sd
FNN	-3.051005	0.1875557
CNN	-2.958988	0.2165333
LSTM	-3.047343	0.2240372

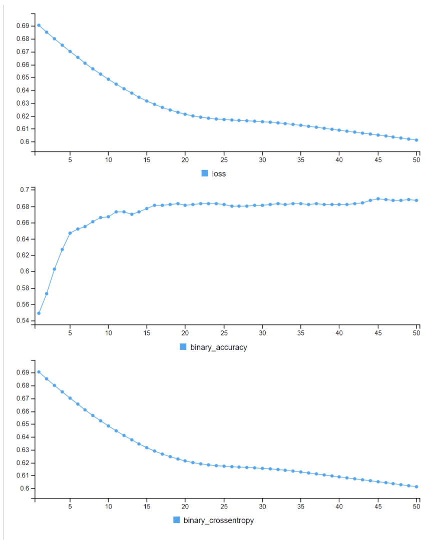
References

Chollet, F. et al. (2015). Keras. https://keras.io.

Farrell, M. H., T. Liang, and S. Misra (2021). Deep neural networks for estimation and inference. *Econometrica 89*(1), 181–213.



(a) The outcome model training



(b) The propensity score training

Figure 5: Errors and accuracy of fits during the learning processes of the nuisance functions using LSTMs

Ghasempour, M., N. Moosavi, and X. de Luna (2024). Convolutional neural networks for valid and efficient causal inference. *Journal of Computational and Graphical Statistics 33*(2), 714–723.

Goodfellow, I., Y. Bengio, and A. Courville (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

Hastie, T., R. Tibshirani, and J. Friedman (2001). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer series in statistics. Springer.

Hinton, G. E., N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov (2012). Improving neural networks by preventing co-adaptation of feature detectors.

Lecun, Y., L. Jackel, L. Bottou, C. Cortes, J. Denker, H. Drucker, I. Guyon, U. Muller, E. Sackinger, P. Simard, and V. Vapnik (1995). *Learning algorithms for classification: A comparison on handwritten digit recognition*, pp. 261–276. World Scientific.

R Core Team (2021). *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing.

Robins, J. M., A. Rotnitzky, and L. P. Zhao (1994). Estimation of regression coefficients when some regressors are not always observed. *Journal of the American Statistical Association 89*, 846–866.

Rosenbaum, P. R. and D. B. Rubin (1983, 04). The central role of the propensity score in observational studies for causal effects. *Biometrika 70*(1), 41–55.

Rubin, D. B. (1974, 10). Estimating causal effects of treatments in randomized and nonrandomized studies. *Journal of Educational Psychology 66*(5), 688–701.

Rubin, D. B. (1980). Randomization analysis of experimental data: The fisher randomization test comment. *Journal of the American Statistical Association 75*(371), 591–593.

Rubin, D. B. (1991). Practical implications of modes of statistical inference for causal effects and the critical role of the assignment mechanism. *Biometrics*, 1213–1234.