Paper ML06

Using R to build artificial neural networks in medical data

Thomas Wollseifen, Syneos Health, Germany

ABSTRACT

Artificial Neural Networks (ANNs) are computer systems that build on existing data-based predictions. ANNs are good for identifying complex patterns in data. They even surpass people at resolving certain problems. The aim of this paper is to present different applications of ANNs with clinical study data. A prediction approach for diabetes is presented based on clinical trial data. Data from the blood glucose level test laboratory of patients diagnosed with diabetes patients is used to train a neural network. The trained neural network is then used to diagnose diabetes in new patients. We will have a look at state of the art so-called convolutional neural networks (CNNs) which can be used to find features in pictures. The statistical software language R with the `neuralnet` library is used to construct differently built ANNs. With the `neuralnet` package, R provides easy-to-use features to build ANNs with different numbers of hidden layers and neurons. The main challenges are choosing the right input parameters, preprocessing the data and normalizing the data to fit into the ANN.

INTRODUCTION

This paper is structured as follows. First, we introduce the topic of neural networks. Using simple examples we show how you can map simple functions with ANNs. After that we go through two examples of diabetes predictions and then look at a state-of-the-art example for categorizing images and how to categorize them with so-called CNNs using the `keras` package in R.

In the first application, a data set for diabetic disorders is considered for implementation. The data set contains 768 records. The inputs considered are age and blood glucose taken at screening and the output is diabetic status (yes or no). Further input parameters such as gender or other pre-existing medical conditions for the ANN can also be tested. The data as classified by medical specialists is used for the training of the neural network. Various settings are made for the topology of the neural network and the output of the predicted data is compared. In the subsequent example we look at another record on diabetes. Here we use 2353 observations where 62% of patients have a diabetes diagnosis and 38% do not. In addition, the data set contains information on age, BMI, blood pressure, blood glucose, Hba1C and smoking habits. These parameters then flow into the ANN.

R's `neuralnet` package makes it easy to train a neural network. Different hidden layers with different numbers of neurons can be created. The process from the training data to the prediction with new test data is shown using examples in R. Before a neural network can be trained, the data must be normalized. The data must also be in numerical form. The examples try different numbers of input parameters for the network. Also, instead of one output neuron, you can create multiple output neurons if the output has more than two categories. A so-called confusion matrix is used to determine the number of true and false positives generated by the predictions.

First, let's look at a brief introduction to neural networks and how they work. We use the R package `neuralnet` [1] to illustrate feed-forward neural networks using some examples. Training data is used to train the neural network. Then we apply test data to the previously trained neural networks and try to predict output values. The test data also contains the predefined output value for comparison. This allows us to determine the accuracy of the prediction in a confusion matrix.

Furthermore, we look at convolutional neural networks (CNNs), which are currently state-of-the-art and show how to recognize certain properties in images. For this we use a record with 70,000 images of the MNIST[1] to train a CNN. RStudio[2] was used for the programming.

INTRODUCTION TO NEURAL NETWORKS

A biological neural network (Fig. 1), consists of nerve cells called neurons. The neurons each have inputs (synapses) and outputs (axons) to which they are linked. The information, i.e. electrical impulses that a neuron receives from its

[1] MNIST - Modified National Institute of Standards and Technology database

[2] RStudio is an integrated development environment for R, a programming language for statistical computing and graphics.

inputs, is processed in a certain way. From this the condition of the neuron is determined. The condition can either be excited or not excited. This state is passed through the outputs to other neurons which in turn determine their condition.

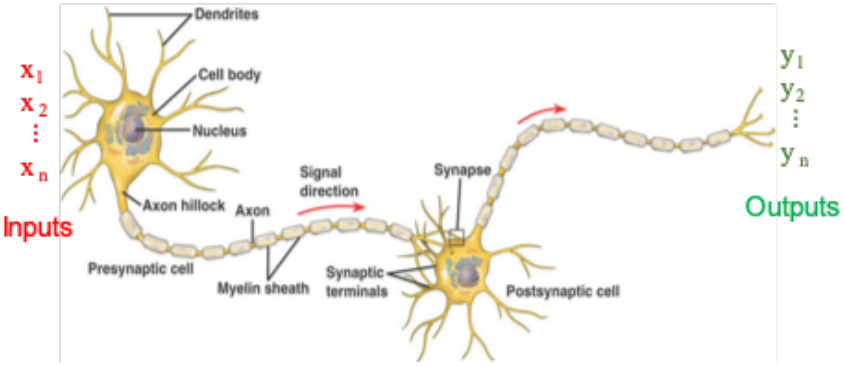


Figure 1: Neuron and axon with signal flow from inputs to dendrites to outputs at the axon ends

ARTIFICIAL NEURAL NETWORKS (ANN)

If you simulate a biological neural network on a computer, you obtain an artificial neural network (ANN) as in Fig. 2. Like the biological neural network, it is made up of neurons that have connections to each other. These neurons are arranged in layers. If every neuron in a layer has connections to all neurons of the following layer, then the net is called fully meshed. From the incoming information, a neuron determines its state by means of a function, usually by summation of the inputs and subsequent scaling with an activation function, e.g. a sigmoid function[3]. The activation function can be seen as a mathematical switch, if a threshold is reached or not reached, the output is either on or off. This state of the neuron can also accept intermediate values. The connections to other neurons can be weighted differently. Additionally, at each layer a bias weight is included. If a specific input is applied to the input layer, then after the adaptation of the states of the individual neurons, the reaction of the neural network is at the output layer. For example, if the artificial neural network has learned a mathematical function, then it provides the result of that for the respective input values in the states of the output neurons.

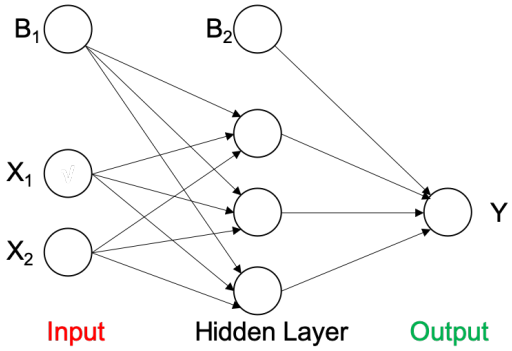


Figure 2: Example of an artificial neural network with two Input neurons (X_1 and X_2), bias weights (B_1 and B_2), a hidden layer, and one output neuron Y

Neurons are connected by links. A link from neuron *j* to neuron *i* serves to propagate the activation a_j from *j* to *i*. Each link also has a numeric weight $w_{j,i}$ associated with it. The weight determines the strength and sign of the connection. Each neuron *i* first computes a weighted sum of inputs:

(1) $in_i = \sum_{j=1}^{n} (W_{j,i} a_j)$ weighted sum of inputs

Then it applies an activation function *g* to this sum to derive the output. The activation function is usually a so-called threshold function. It could be a sigmoid function or logistic function for example. The sigmoid function is differentiable, which is important for the weight-learning algorithm.

[3] A sigmoid function is also called logistic function: $f(x) = \frac{1}{1+e^{-x}}$

Various activation functions are possible, for example a sigmoid function or logistic-sigmoid function has been used in the past. In recent years it has been shown that a ReLu[4] function shows very good results in deep learning. The sigmoid function and the ReLU functions are displayed in Fig. 3.

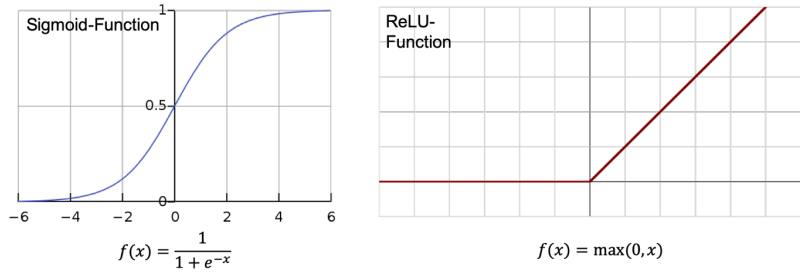


Figure 3: Activation functions (source Wikipedia)

(2) $a_i = g(in_i) = g\left(\sum_{j=1}^{n}(W_{j,i}a_j)\right)$ activation function

The following figure Fig. 4 shows a simple mathematical model of a single neuron.

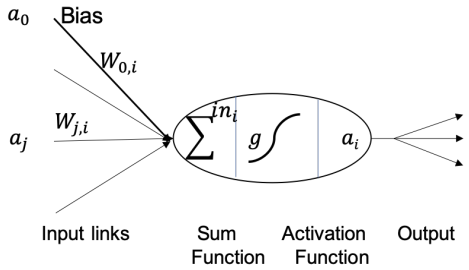


Figure 4: Mathematical model of a neuron

The unit's output activation is a_i as given in formula (2), where a_j is the output activation of unit j and $W_{j,i}$ is the weight on the link from unit j to this unit. When the output for each of the neurons in a layer is calculated the result of each neuron is fed to the next layer and so on until the output neuron is reached (feed-forward network). Then the error (difference between the calculated and currently predicted output) is calculated. Thereafter, the error is back-propagated to the previous layer and so on and the weights are re-adjusted. Then a new iteration starts. The back-propagation[5] algorithm is shown in Figure 5.

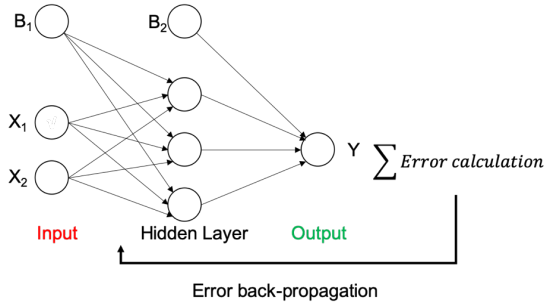


Figure 5: Back-propagation

[4] The rectifier function is an activation function (also called "ramp-function"). A unit employing the rectifier is also called a rectified linear unit (ReLU).

[5] The name of the algorithm results from the back-propagation of the error.

The back-propagation algorithm runs in the following phases:

- An input pattern is created and propagated forward through the network.
- The output of the network is compared to the desired output. The difference between the two values is considered a fault of the network.
- The error is now propagated back to the input layer via the output layer. The weights of the neuron connections are changed depending on their influence on the error. This guarantees an approximation to the desired output when the input is re-applied.

STRUCTURE OF ARTIFICIAL NEURAL NETWORKS

ANNs can have different structures. There are feed-forward networks, where one layer is always connected to the next higher layer. In addition, there are networks in which connections are allowed in both directions. The appropriate network structure is usually found using trial and error, which can be supported by evolutionary algorithms and error feedback. In this work we will only look at feed-forward networks.

Single-layer feed-forward network

Single-layer networks with the feed-forward property are the simplest structures of artificial neural networks. They only have one output layer. The feed-forward property states that neuron outputs are directed only in the processing direction and cannot be returned by a recurrent edge (acyclic, directed graph).

Multilayer feed-forward network

In addition to the output layer, multi-layer networks also have hidden layers whose output, as described, are not visible outside the network. Hidden layers enhance the abstraction of such networks. Later we will show some examples of multilayer feed-forward networks.

Recurring network

In contrast, recurrent networks also have reverse feedback edges (feedback loops) and thus contain feedback. Such edges are then often provided with a time delay so that, in a stepwise processing, the neuron outputs of the past unit can be re-entered as inputs. These feedbacks allow a network to have dynamic behavior and endow it with memory.

In the next section we will first introduce the R's `neuralnet` package, which is a simple tool to start with ANNs. Then we'll look at simple examples and later more complex ANNs in the field of diabetes.

R'S NEURALNET PACKAGE

The `neuralnet` package in R is built to train multi-layer perceptrons (MLPs)[6] in the context of regression analyses. With R's `neuralnet` package multi-layer perceptrons can be mapped, which are well suited for the modeling of functional relationships. The underlying structure of an MLP is a directed graph, it consists of vertices and directed edges, called neurons and synapses in this context. We briefly introduced an MLP in the previous section. The neurons are organized in layers that are normally completely connected by synapses. In `neuralnet`, a synapse can only be connected to subsequent levels. The input layer consists of all covariates in separate neurons and the output layer consists of the response variables. The intervening layers are called hidden layers. The input layer and the hidden layers contain a constant neuron, which is the bias.

`Neuralnet` focuses on supervised learning algorithms. These learning algorithms are characterized by the usage of a given output that is compared to the predicted output and by the adaptation of all parameters according to this comparison. The parameters of a neural network are its weights. All weights are usually initialized with random values drawn from a standard normal distribution. During an iterative training process, the following steps are repeated:

- calculate the output for the given inputs and the current weights
- an error function like sum of squared errors is calculated
- the difference between predicted and observed output is calculated
- all weights are adapted according to the rule of a learning algorithm

This process stops if a pre-specified criterion is fulfilled, e.g. if the derivatives of the error function regarding the weights are smaller than a given threshold. A back-propagation algorithm is applied.

[6] A multilayer perceptron (MLP) is a class of feed-forward artificial neural networks. An MLP consists of at least three layers of nodes: an input layer, a hidden layer and an output layer.

The back-propagation algorithm is based on modifying the weights of a neural network to find a local minimum of the error function. Therefore, the gradient of the error function (dE / dw) with respect to the weights is calculated to find a root. In particular, the weights are changed in the opposite direction of the partial derivatives until a local minimum is reached.

In `neuralnet` different learning algorithms are possible. It provides the possibility to choose back-propagation and resilient back-propagation with or without weight backtracking. All algorithms try to minimize the error function by adding a learning rate to the weights going into the opposite direction to the gradient. `Neuralnet` has an interesting function that allows the neural network to be plotted. We can use the function *plot()* and pass the output object (in this case the ANN) to it. An example of the drawn neural network is shown in Fig. 6.

Starting with a simple example:

EXAMPLE OF AN ANN WITH NEURALNET

In the first example we build a simple neural network that is supposed to compute the square root.

Input	Square root expected	Neural net output predicted
0	0	0.7143873
1	1	1.0243201
4	2	2.0041862
9	3	3.0001876
16	4	4.0006444
25	5	4.9958750
36	6	6.0030675
49	7	7.0034939
64	8	7.9974201
81	9	9.0019381

Table 1 Expected and calculated result by `neuralnet`

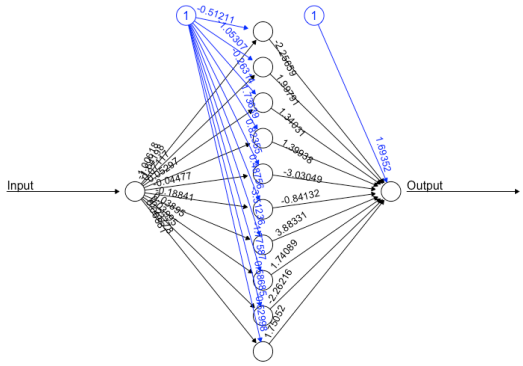


Figure 6 Neuronal network with 10 hidden neurons to calculate the square root function

We generate 50 random numbers, evenly distributed between 0 and 100, and store them as R *data frame*[7]. This data frame will be used as training dataset of the neural network. The neural network is generated with the following `neuralnet` command.

```
nn.sqrt <- neuralnet(Output~Input,trainingdata, hidden=10, threshold=0.01)
```

The values calculated by the neural network on the test data can be calculated with the `predict` function.

```
net.results <- predict(net.sqrt, testdata)
```

In the previously trained value range of the training data, the ANN provides a very good approximation to the actual function. The result of the predicted values is shown in Tab. 1. If you want to use the newly generated neural network to calculate square roots outside the trained range, the neural network will no longer provide reliable results. This effect is shown in the following figures (Fig. 7 and 8). The green dashed line shows the expected square root and the red line shows the predicted square root by the neural network. The range for which the ANN was trained is shown between the blue, dashed, vertical lines.

[7] A data frame is used for storing data tables. It is a list of vectors of equal length. A data frame is comparable to a SAS data set.

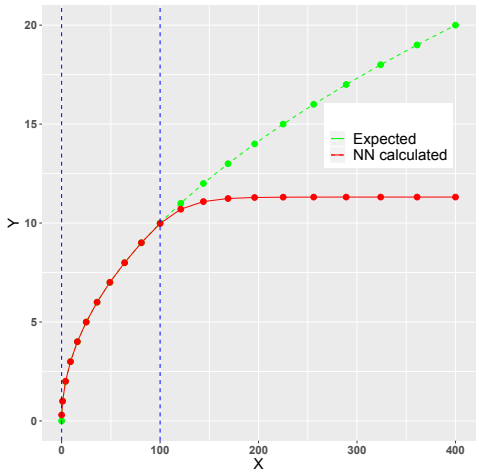


Figure 7: Difference between expected and ANN calculated square root function

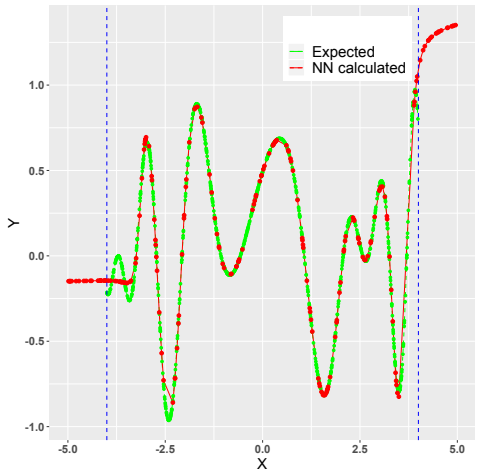


Figure 8: Sine/Cosine function, expected and ANN calculated results

This effect is also clear in Figure 8. Here a combination of sine / cosine functions was simulated and calculated with an artificial neural network. Outside the trained value range, the neural network does not provide reliable values. However, this is an expected effect because you train neural networks for specific data.

EXAMPLE 1: PIMA INDIANS DIABETES

Let's look at an example of training neural networks in the field of diabetes. The Pima are a group of native Americans living in Arizona. A genetic predisposition allowed this group to survive normally on a diet poor of carbohydrates for years. In the recent years, a sudden shift from traditional agricultural crops to processed foods, together with a decline in physical activity, has made them develop the highest prevalence of type-2 diabetes and for this reason they have been the subject of many studies.

The data set we use for the ANN contains data from 768 women with 8 characteristics [2], in particular:

	FEATURE	MEAN ± STANDARD DEVIATION
COVARIATES	1. Number of times pregnant	3.8 ± 3.4
	2. Plasma glucose concentration (over 2 hours in an oral glucose tolerance test)	120.9 ± 32.0
	3. Diastolic blood pressure (*mmHg*)	69.1 ± 19.4
	4. Triceps skin fold thickness (*mm*)	20.5 ± 16.0
	5. 2-Hour serum insulin (*mu U/ml*)	79.8 ± 115.2
	6. Body mass index (kg/m^2)	32.0 ± 7.9
	7. Diabetes pedigree function (DPF)	0.5 ± 0.3
	8. Age (*years*)	33.2 ± 11.8
OUTCOME VARIABLE	Diabetes (outcome variable) Yes=1 / No=0	N=268 (35%) diabetes N=500 (65%) not diabetes

Table 2 Covariates and outcome variable of Pima Indians diabetes dataset

For each person was indicated whether diabetes was diagnosed (1 = 'yes') or not (0 = 'no'); this is the outcome variable. Diabetes was diagnosed in 35% of the population and no diabetes was found in 65%.

With the help of a correlation matrix, you can already verify similarities and then focus on parameters with the highest correlations and use them for the neural network. In the correlation matrix, it can be seen that glucose and BMI have slightly higher correlation coefficients to diabetes than the others, and are probably of greater importance for the ANN. In fact, the correlation coefficient for these two parameters is slightly higher than 0.5, so you really cannot speak of a correlation.

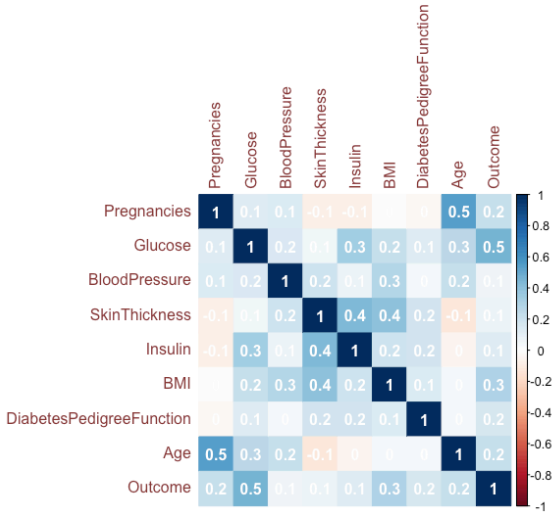


Figure 9: Correlation matrix of all covariates and outcome variable (diabetes)

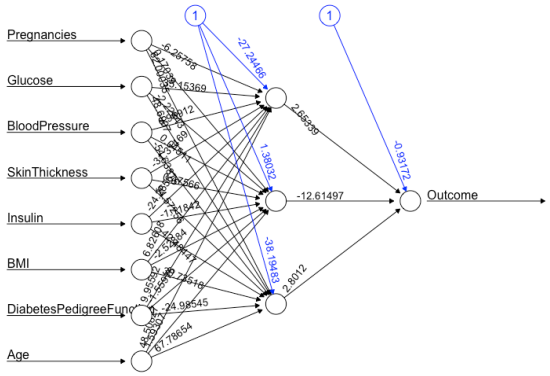
In order to improve the results, the data set is normalized before the ANN is trained.

```
normalize <- function(x) {
  return ((x - min(x)) / (max(x) - min(x)))
}
diabetes.df_norm <- as.data.frame(lapply(diabetes.df, normalize))
```

The normalize function normalizes the data in the range of 0 to 1. We have used 90% of the data for training the ANN and 10% for testing the ANN. The sample function randomly draws records from the diabetes data set for the training set and the subsequent test set without replacement.

```
index <- sample(1:nrow(diabetes.df_norm),round(0.90*nrow(diabetes.df_norm)))
trainset <- diabetes.df_norm[index,] # 90% of data for training the neural network
testset <- diabetes.df_norm[-index,] # 10% of data for testing the neural network
```

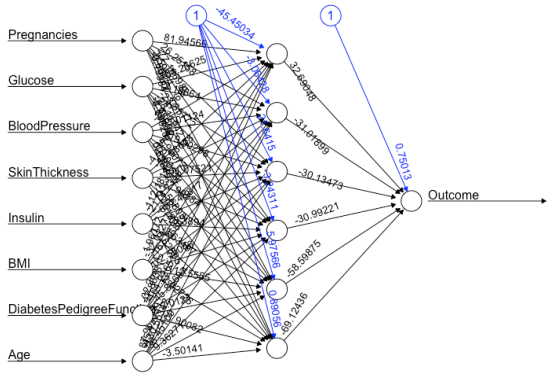
With the given covariables the ANN can now be trained. As found above with the correlation matrix, different covariates may be used for the ANN, possibly having a greater impact on predictability. We also tested whether different numbers of hidden neurons affect the accuracy of the prediction. The following figures show different configurations of ANNs that are constructed with different numbers of hidden neurons. We have calculated the accuracy of the actual vs. predicted categorization. This is shown in a confusion matrix, also known as an error matrix. Each row of the matrix represents the instances in a predicted class while each column represents the instances in an actual class (or vice versa). The false / positive rate shows the ratio of correctly predicted diabetes to total prediction. Furthermore, we have calculated the mean square error (MSE). If it is closer to zero, the error is smaller.



Actual		prediction	
		0	1
	0	46	9
	1	9	13

False/positive rate = 0.77
MSE = 0.18

Figure 10 ANN with 3 hidden neurons / confusion matrix



Actual		prediction	
		0	1
	0	46	9
	1	5	17

False/positive rate = 0.82
MSE = 0.17

Figure 11 ANN with 6 hidden neurons / confusion matrix

We also created just one ANN with the variables glucose and BMI. However, this showed only an accuracy of 73% and an MSE of 0.18 for 4 hidden neurons.
Overall, an ANN with 8 covariates and 6 hidden neurons achieved an accuracy of 82% of diagnosis with 10% of test data. In other works, using different approaches similar or slightly higher accuracies were achieved.
The accuracy in ANNs certainly depends on the data volume and what influence the parameters have on the output variable. In this context we now look at another example in the field of diabetes diagnosis.

EXAMPLE 2: DIABETES DATA

In the next example, we have a data set of 2353 observations with a diagnostic variable (diabetes 62%, no diabetes 38%) and 8 covariates (blood glucose, age, BMI, systolic blood pressure, diastolic blood pressure, sex, HBa1C, smoking habits). First, we again set up a correlation matrix, which gives a nice overview of the possible correlations. It shows that the covariable HBa1C (r = 0.9) probably has a larger correlation to the diagnosis of diabetes. The other covariates seem to have a smaller impact on diabetes in the order of glucose, age, BMI and smoking. However, systolic and diastolic blood pressure correlate to each other, but apparently the correlation to diabetes is low.

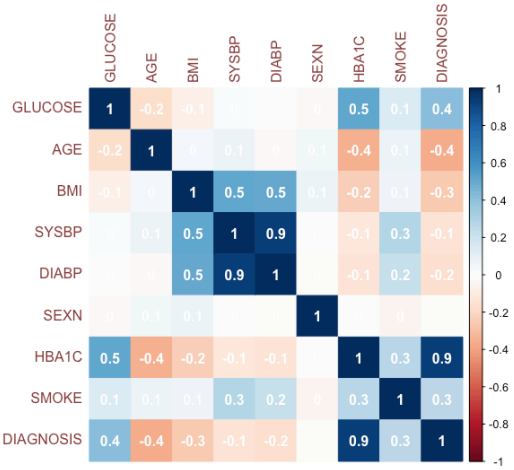


Figure 12 Correlation matrix of all covariates and outcome variable (diagnosis = diabetes)

We can also examine the data using histograms to see how the data in each variable is distributed in terms of the diagnosis variable. For example, we expect that the glucose value in the diabetes group is slightly higher than in the non-diabetes group. Likewise, the BMI in the diabetes group is slightly higher than in the non-diabetes group.

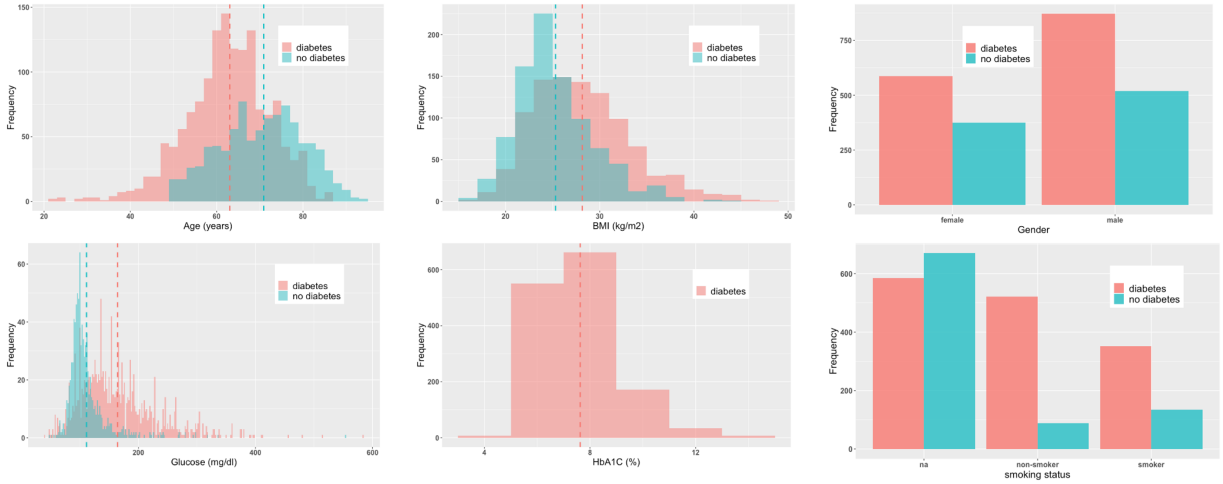
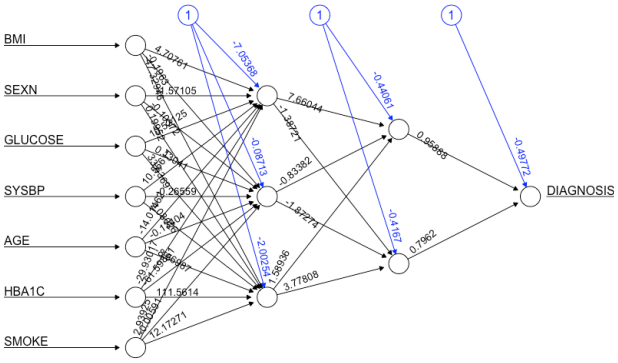
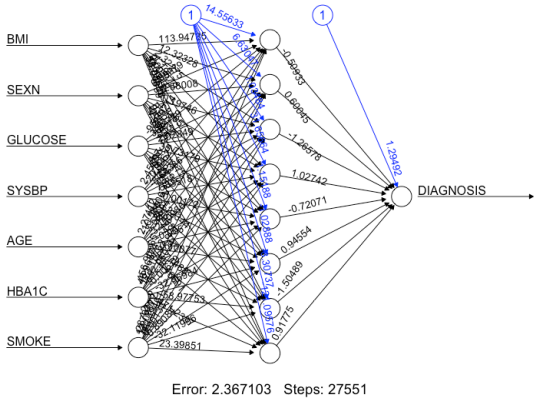


Figure 12 Histograms of covariables per diabetes status

In the first step, we generated an ANN with 7 covariables and 3×2 hidden neurons. This gave a false positive rate of 99% and a MSE of 0.05 which is a very good result. In the second approach, we used an ANN with 7 covariables and 8 hidden neurons. The accuracy was even slightly improved. The results are shown in Figure 14 and 15. Underneath the graphics you can also see the corresponding confusion matrices with the respective false / positive rate.





Actual		prediction 0	prediction 1
	0	97	0
	1	2	136

False/positive rate = 0.99
MSE = 0.005

Figure 14 ANN with 3x2 hidden neurons / confusion matrix

Actual		prediction 0	prediction 1
	0	96	1
	1	0	138

False/positive rate = 0.996
MSE = 0.004

Figure 15 ANN with 8 hidden neurons / confusion matrix

So far we have considered simple ANNs. In the following chapter we take a look at currently used methods of deep learning in R. We will have a look at the R's `keras` package which can introduce convolutional neural networks and the TensorFlow framework.

INTRODUCTION TO DEEP-LEARNING IN R

Here we cover the training of deep learning algorithm for binary classification of malignant/benign cases of breast cancer. This will be possible using a machine learning framework. We use convolutional neural networks in R. Then we look at deep-learning algorithms for recognizing certain properties or categories in images.

CONVOLUTIONAL NEURAL NETWORKS (CNNS)

Convolutional Neural Networks[8] (CNNs) are a special kind of multi-layer neural network. They are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels at one end to class scores at the other. And they still have a loss function on the last (fully-connected) layer. However, convolutional neural network architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network. Therefore, they can recognize patterns with extreme variability (such as handwritten characters or features in images), and with robustness to distortions and simple geometric transformations.

Here are some features of convolutional neural networks:

- Similar to feed-forward neural networks.
- Convolutional neural networks (CNN) model can be applied to visual recognition tasks.
- The architecture of a CNN is designed to take advantage of the grid structure of data.
- Hierarchy of representations with increasing level of abstraction.
- Each stage is a kind of trainable feature transform.

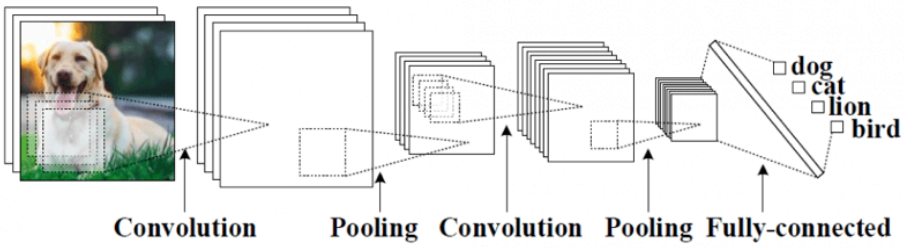


Figure 16 Convolutional Neural Network (CNN)[9]

There are three main operations in the CNN:

1. Convolution
2. Pooling or Sub Sampling
3. Classification (Fully Connected Layer)

These operations are the basic building blocks of every convolutional neural network. CNNs derive their name from the *convolution operator*. The primary purpose of convolution within CNN is to extract features from the input image. Convolution preserves the spatial relationship between pixels by learning image features using small squares of input data. We will not go into the mathematical details of convolution here, but will try to understand how it works with images.

Pooling (also called subsampling or down sampling) reduces the dimensionality of each feature map but retains the most important information. Spatial Pooling can be of different types: Max, Average, Sum etc.

In the case of max pooling, we define a spatial neighborhood (for example, a 2×2 window) and take the largest element from the rectified feature map within that window. Instead of taking the largest element we could also take the average (average pooling) or sum of all elements in that window. An example of max pooling is given in Fig. 17.

[8] The name "convolutional neural network" indicates that the network employs a mathematical operation called convolution. Convolution is a special kind of linear operation.

[9] https://www.ayasdi.com/blog/artificial-intelligence/using-topological-data-analysis-understand-behavior-convolutional-neural-networks/

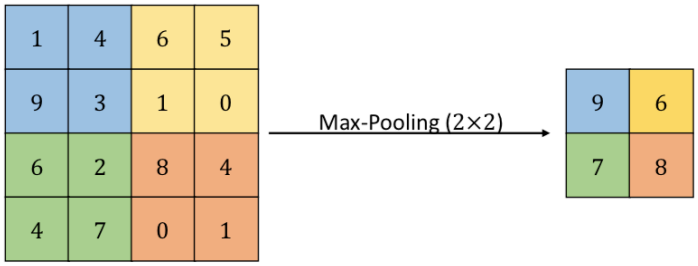


Figure 17 Pooling step in a CNN, reduction to essential information

Pooling replaces the output of the network at a particular location with summary statistics of the neighboring outputs. It reduces the dimension and shows essential information of a graphic.
At the end of the CNN a fully connected layer is added. The output from the convolutional and pooling layers represent high-level features of the input image. The purpose of the fully connected layer is to use these features for classifying the input image into various classes based on the training dataset.
In the following example, we apply a CNN to breast cancer data with various characteristics that characterize the cancer as malignant or benign.

TRAINING A NEURAL NETWORK TO IDENTIFY TYPES OF BREAST CANCER WITH CNNS

Breast cancer is the presence of fast-growing breast cells that eventually form a tumor. The tumor is malignant when the cells are able to grow into the surrounding tissue or spread (metastases) to distant body regions.
We now use a record from the University of California, which provides information on breast cancer cases. The dataset contains 569 entries with 30 covariates for the appearance of the examined tissue. In addition, a classification variable (diagnosis) with the entries malignant or benign is present. We generate again training data and test data for the future with an convolutional neural network. We can use the CNN functionality by including the package `keras` in R.

```
library(keras)
```

`Keras` is a high-level neural networks API (application programming interface) developed with a focus on enabling fast experimentation. This helps us to create the layered neural network very easily. It includes a variety of layers to include in the convolutional neural network. `Keras` provides a vocabulary for building deep learning models that is simple, elegant, and intuitive. With `keras` we can use CNNs in R.

First, let's look at correlations between the different variables of the breast cancer data set. Our data set contains information on the shape, texture and other properties of tumors. They can be found in the following correlation plot including the diagnostic variable at the bottom of the table. The dark blue colors (or dark red colors) indicate a higher correlation.

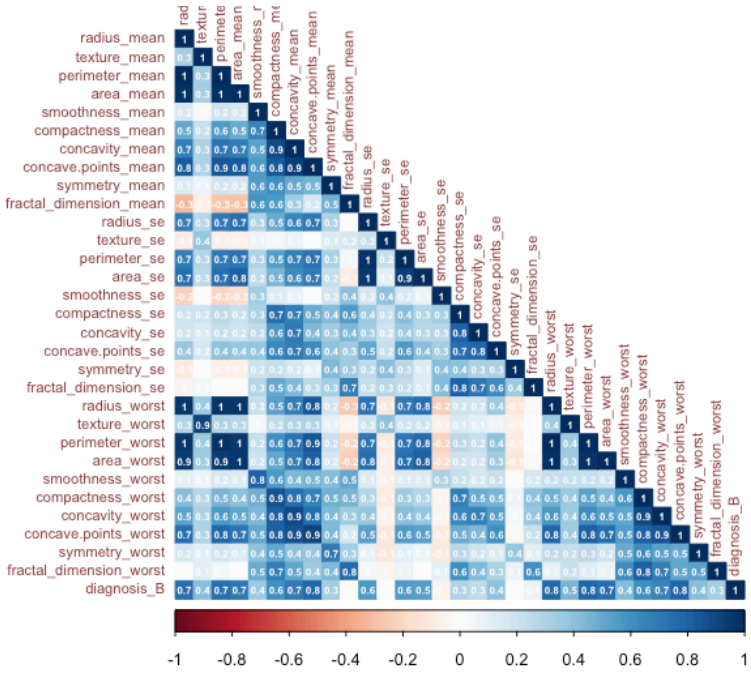


Figure 18 Correlation matrix of all covariates and outcome variable (diagnosis = malignant / benigne)

The network design is assigned using the following R statements. As previously mentioned, it consists of different layers.

```
model <- keras_model_sequential()
model %>%
# Input layer
  layer_dense(units = 256, activation="relu", input_shape =  ncol(X_train)) %>%
  layer_dropout(rate = 0.4) %>%
# Hidden layer
  layer_dense(units = 75, activation = "relu") %>%
# Output layer
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 2, activation = "sigmoid")

model %>% compile(optimizer = 'adam',loss = 'sparse_categorical_crossentropy',
                  metrics = c('accuracy'))
# Fit the model
model %>% fit(X_train, y_train, epochs = 12, batch_size=5, validation_split=0.2)
```

At each layer a different activation function is used. Besides the Sigmoid function, we will also use the so called Rectified Linear Unit function or ReLU which in essence helps the algorithm to learn more quickly and reduce the likelihood of the gradient to vanish while optimizing the model. To find the optimal weights a so called ADAM optimization algorithm was used. Other optimization methods are also possible. In addition, we will be dropping some of the points in order to avoid overfitting of the model by introducing some noise into the learning process. This is done with the `layer_dropout` statement. Although this depends on the characteristics of the data set, it is common to use between 20% and 40% for the dropout. During the training period, a neuron with the probability p is temporarily "dropped" or deactivated at each iteration. This means that all inputs and outputs to this neuron are disabled on the current iteration. The failing neurons are resampled at each training step with probability p so that a failing neuron can be active again in the next step. Dropout is used to prevent the CNN from overfitting. Figure 19 shows the effect of drop-outs on the neural network.

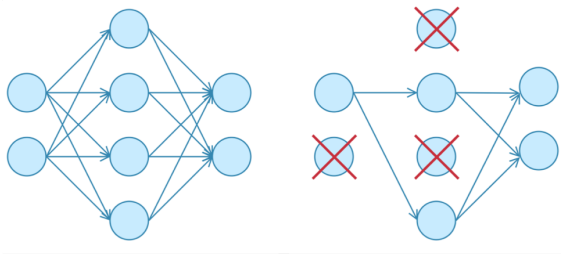


Figure 19: Effect of the drop-out of nodes on the network

In the last statement the model is trained with the `fit` statement for 12 epochs using batches of 5 training records for each step. The result of the training epochs is shown in the following performance diagram (Fig. 20). We achieved a high accuracy of over 99%.

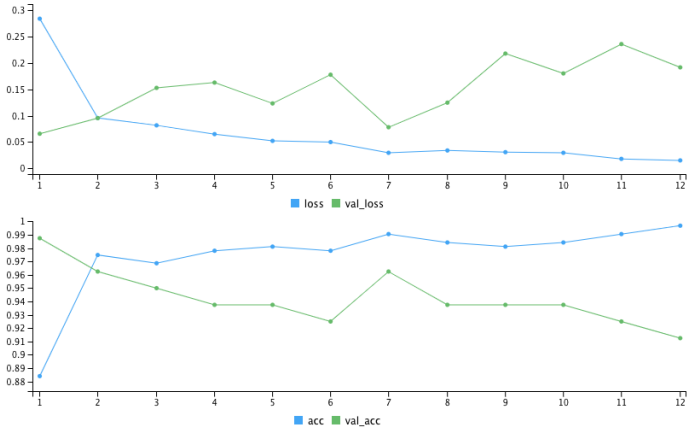


Figure 20 Performance diagram of the training epochs

The confusion matrix showed also a high accuracy.

		Prediction	
		benign	malignant
Actual	benign	103	2
	malignant	3	62

False/positive rate = 0.97

We also tried to use the `neuralnet` package and apply an ANN to the breast cancer dataset with the same training/test data. We achieved an accuracy (in the false/positive rate) of 92% compared to the CNN with 97% which is significant improvement.
In the next section, we'll look at the CNNs that can be used for image characterization.

TRAINING A CONVOLUTIONAL NETWORK FOR IMAGE DETECTING FEATURES

We will train a CNN to categorize pictures of garments. We used the fashion MNIST dataset [3], which contains 70,000 grayscale images in 10 categories.

```
class_names <- c('T-shirt/top','Trouser','Pullover', 'Dress', 'Coat', 'Sandal',
                 'Shirt','Sneaker','Bag', 'Ankle boot')
```

The picture (Fig. 21) show single garments in low resolution (28 × 28 pixels). The dataset is already accessible within the `keras` package and can be downloaded directly.

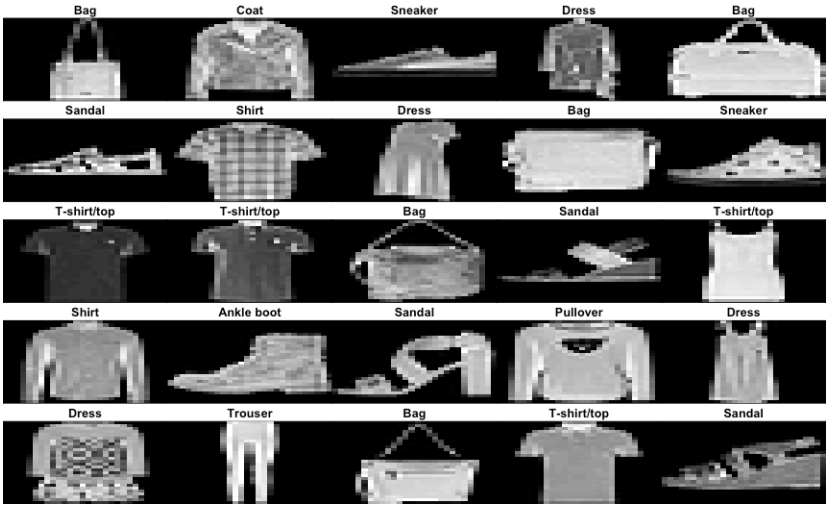


Figure 21 Example of MNIST fashion dataset with categories of clothing

The idea now is to design a convolutional neural network with `keras` to categorize new pictures of fashion. The MNIST fashion dataset is the “Hello, World” of machine learning programs for computer vision. We will use 60,000 images to train the network and 10,000 images to evaluate how accurately the network learned to classify images. We can read the data into R using the following statements:

```
fashion_mnist <- dataset_fashion_mnist()
c(train_images, train_labels) %<-% fashion_mnist$train
c(test_images, test_labels) %<-% fashion_mnist$test
```

The images are stored altogether in a matrix and the labels (numbers from 0 to 9) in a vector. This is necessary for the CNN model since it needs matrices and vectors as input instead of a data frame.
The image data needs to be preprocessed in order to fit into the neural network. The pixel value range from 0 to 255. We scale these values to a range of 0 to 1 before feeding to the neural network model. It’s important that the training set and the testing set are preprocessed in the same way.

```
train_images <- train_images / 255
test_images <- test_images / 255
```

Now we can build the neural network. It consists of different layers that are chained together.

```
model <- keras_model_sequential()
model %>%  layer_flatten(input_shape = c(28, 28)) %>%
           layer_dense(units = 128, activation = 'relu') %>%
           layer_dense(units = 10, activation = 'softmax')
```

The first layer in this network, `layer_flatten`, transforms the format of the images from a 2d-array (of 28 by 28 pixels), to a 1d-array of 28×28 = 784 pixels. This is just a transformation layer.
After the pixels are flattened, the network consists of a sequence of two dense layers. These are densely-connected, or fully-connected, neural layers. The first dense layer has 128 nodes. The last layer is a 10-node *softmax* layer —this returns an array of 10 probability scores that sum to 1. Each node contains a score that indicates the probability that the current image belongs to one of the 10 digit classes.
In the next step the model is compiled with the `compile` statement. We add a few more settings. The loss function measures how accurate the model is during training. This function will be minimized. The optimizer sets how the model

is updated based on the data it sees and its loss function. In this case it uses the so-called ADAM algorithm. The metrics option monitors the training and testing steps.

```
model %>% compile(optimizer = 'adam', loss = 'sparse_categorical_crossentropy',
                  metrics = c('accuracy'))
```

In the training step we feed the train images and corresponding training labels to the model. The training is started with the `fit` method.

```
model %>% fit(train_images, train_labels, epochs = 8)
```

In R, you can follow the process of each training epoch in the console, and the `fit` method also creates an image of the process being added at the same time.

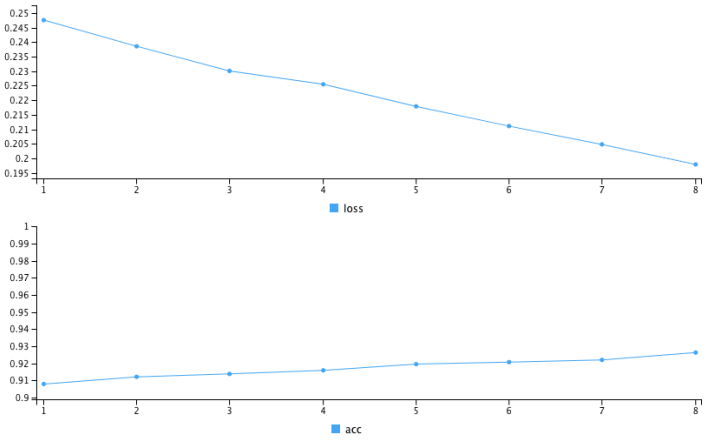


Figure 22 Performance diagram of the training epochs

Finally, we achieved an accuracy of 93%. The following image shows examples of the tested images along with the predicted categories. The correct labels are displayed in green while the mis-predicted categories are displayed in red.

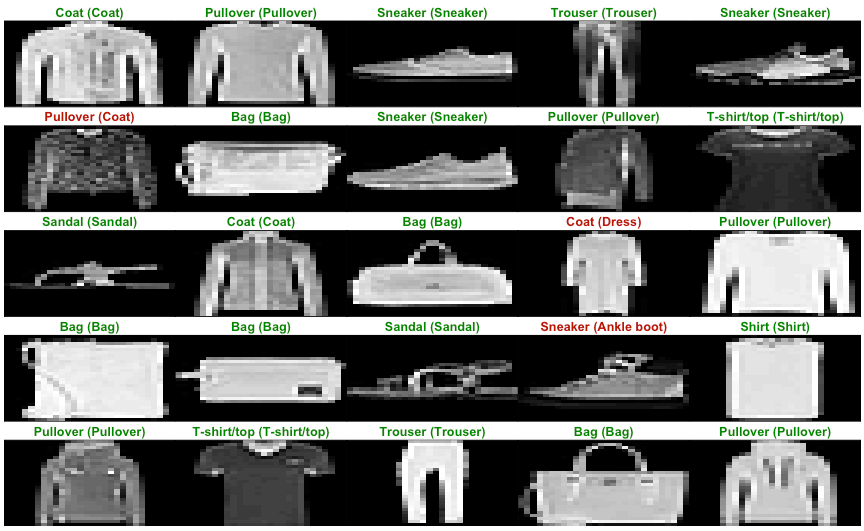


Figure 23 Performance diagram of the training epochs

We can for example grab a picture from the test set and predict the label by using the `predict` method.

```
number_test_image<-1022
img <- test_images[number_test_image, , , drop = FALSE]
predictions <- model %>% predict(img)
predictions
prediction <- predictions[1, ] - 1
class_names[which.max(prediction)]
```

In this case we select image no. 1022 for the prediction. We obtain a list of the 10 labels and choose the label with the highest prediction value of the category and then we return it in the label format, which is 'pullover'.

```
          [,1]         [,2]      [,3]         [,4]      [,5]         [,6]      [,7]         [,8]         [,9]         [,10]
[1,] 0.0003279748 8.492876e-11 0.9668224 1.83288e-08 0.01744946 1.467221e-15 0.01540017 6.208517e-20 1.076891e-08 1.039985e-12
"Pullover"
```

Figure 24 Image no.1022 with predicted label "Pullover"

CONCLUSION

In this work, we got to know R's `neuralnet` package with some examples. With `neuralnet` you can render multilayer perceptrons. Although these have been known since the 1970s and have been used since then, there has been no breakthrough in machine learning until recently. Improvements have been made e.g. achieved with Convolutional Neural Networks in image recognition in recent years. Similarly, the open source framework TensorFlow, which was developed especially for image recognition, brought new achievements in the field of image recognition. Big internet companies like Google use this when classifying images. With R's package `keras` Tensorflow is very easy to integrate and the structure of CNNs resembles a sandbox. Multiple layers are easy to integrate and complex models can be created. Overall, I would continue to work with CNNs in R, as they are currently state of the art. Particularly in the area of image recognition, we have seen from the example of the MNIST fashion data set that classifications can be carried out very simply and with good accuracy. Of course, this can easily be applied to medical images. There are some advantages of CNNs compared to ANNs:

- robustness
- less storage space required
- easier and better training

The feature extraction reduces the amount of data and space considerably. CNNs are not fully linked in the first layers. Only in the last layer is a fully-linked network added for classification.

Especially with `keras` it is very easy to build complex CNNs with multiple layers. You can always try different configurations and see how the new network is trained and how accurate the network's predictions are.

In addition to the R packages introduced in machine learning, R also offers other packages. Some of these are the `h2o`, `deepnet` or `mxnet` packages. We will certainly see further improvements in the field of machine learning in the

future. This paper has given a little insight into creating neural networks and applying them to your data with very little effort.

REFERENCES

[1] Günther F. and Fritsch S.: neuralnet: Training of Neural Networks, The R Journal Vol. 2/1, June 2010

[2] Pima Indians, data https://www.kaggle.com/kumargh/pimaindiansdiabetescsv

[3] MNIST fashion data https://www.kaggle.com/zalando-research/fashionmnist

ACKNOWLEDGMENTS

I would like to thank Rowland Hale (Senior Principal Statistical Programmer, Syneos Health) for his great support and review of this manuscript.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Thomas Wollseifen
Syneos Health Germany GmbH
Stefan-George-Ring 6
81929 München, Germany
Email: thomas.wollseifen@syneoshealth.com
Web: syneoshealth.com

Syneos Health | Shortening the distance from lab to life.™