Introduction to Artificial Neural Networks - Neurons

Module 2 | Chapter 5 | Notebook 3

In this chapter you'll get to know neuronal networks: By the end of this lesson you will know:

- What artificial neurons are and how they work.
- How to put artificial neurons into practice.
- What is the relationship between logistic regression and artificial neurons.

Artificial neurons

Scenario: You work for the company *Lemming Loans Limited*. The company provides loans to private individuals. Investors indicate how much money they want to make available and the system pools the money from different investors and forwards this money to people who want to take out a loan. The people who take out a loan often have a low credit rating, which is why they aren't getting the loans from a bank in the traditional way. Particularly risky loans get an interest rate of over 14%. These loans are internally classified as problem loans and require more attention from *Lemming Loans Limited*.

Previously, a service provider calculated the interest rate for each loan. The services of the external provider are now going to be taken over step by step by internal departments. This saves costs and makes the assessment more transparent for *Lemming Loans Limited*. The first task is to automatically divide the loans into problem loans and normal loans. It is therefore a supervised learning binary classification problem.

You already prepared and stored the data in a pickle in the last exercise. We have separated the training data and validation data. Let's import it quickly

In [1]:
```python
import pandas as pd
features_train = pd.read_pickle('features_train.p')
features_val = pd.read_pickle('features_val.p')
target_train = pd.read_pickle('target_train.p')
target_val = pd.read_pickle('target_val.p')

features_train.head()
print(target_train.value_counts(normalize=True))
print(target_val.value_counts(normalize=True))
```

```
0.0    0.525996
1.0    0.474004
Name: problem_loan, dtype: float64
0.0    0.525476
1.0    0.474524
Name: problem_loan, dtype: float64
```

You can see that our target variable, both in the training and validation set, is fairly evenly distributed. Our features are organized as a `DataFrame` with 77 columns. The row names match those of the original data set in *loans.db*. They were just mixed in the separation into training and validation data. Each row corresponds to one loan application. The following data dictionary describes the data:

Column number	Column name	Type	Description
0-6	`'grade'`	categorical	Assigned credit score grade by external service provider (one-hot encoded)
7-41	`'sub_grade'`	categorical	sub-grade of assigned credit score grade by external service provider (one-hot encoded)
42-47	`'home_ownership'`	categorical	Housing situation (one-hot encoded)
48-50	`'verification_status'`	categorical	Indicates whether income has been verified or not (one-hot encoded)
51-60	`'1d_zip'`	categorical	first digit of the borrower's zip code (one-hot encoded)
61-67	`'purpose'`	categorical	Reason for loan (one-hot encoded)
68	`'funded_amnt'`	continuous (`float`)	Amount loaned in USD
69	`'term'`	continuous (`int`)	Number repayment installments in months (36 or 60)
70	`'annual_inc'`	continuous (`float`)	annual income in USD
71	`'total_acc'`	continuous (`int`)	Total number of borrower's loans
72	`'percent_bc_gt_75'`	continuous (`float`)	Proportion of credit cards up to 75% of their limit
73	`'total_bc_limit'`	continuous (`float`)	total credit card limit
74	`'revol_bal'`	continuous (`float`)	Open credit card amounts
75	`'emp_length_num'`	categorical (ordinal)	Length of borrower's current employment in years when taking out the loan
76	`'unemployed'`	categorical	Whether borrower was presumably unemployed (`1`) or not (`0`). This feature was generated in *Presenting and Preparing Loan Data*

Management at *Lemming Loans Limited* has learned that artificial neural networks (**ANNs** for short) are currently a hot topic. They want to take advantage of this and advertise to their customers that they are taking the latest technological advances into account in their decision-making processes, thus going the extra mile in terms of sustainability and security. That is why they would like you to create a prediction model based on artificial neural networks.

But before we look at the networks themselves, we want to look at what **artificial neurons** (**ANs** for short) are.

To understand this, we'll make a short detour into the field of neurobiology. That's right! And this is not surprising. A lot of technological advances use nature as a model and inspiration. Think of airplanes with birds as a clear example, or the famous lotus effect, which was copied from the lotus plant. Artificial neurons are modelled on brain cells - neurons. Roughly speaking, neurons are responsible for the signal communication between parts of the body. They ensure that things function properly. When we move our arms, for example, millions of neurons are involved in this process to communicate signals between the brain and the arm.

Neurons constantly form connections with each other in large collections of neurons - neural networks. But before we go into the networks, in this lesson we'll look at how a single neuron works and use this as an introduction to artificial neurons.

Now let's look at a neuron in the human brain in the following figure.

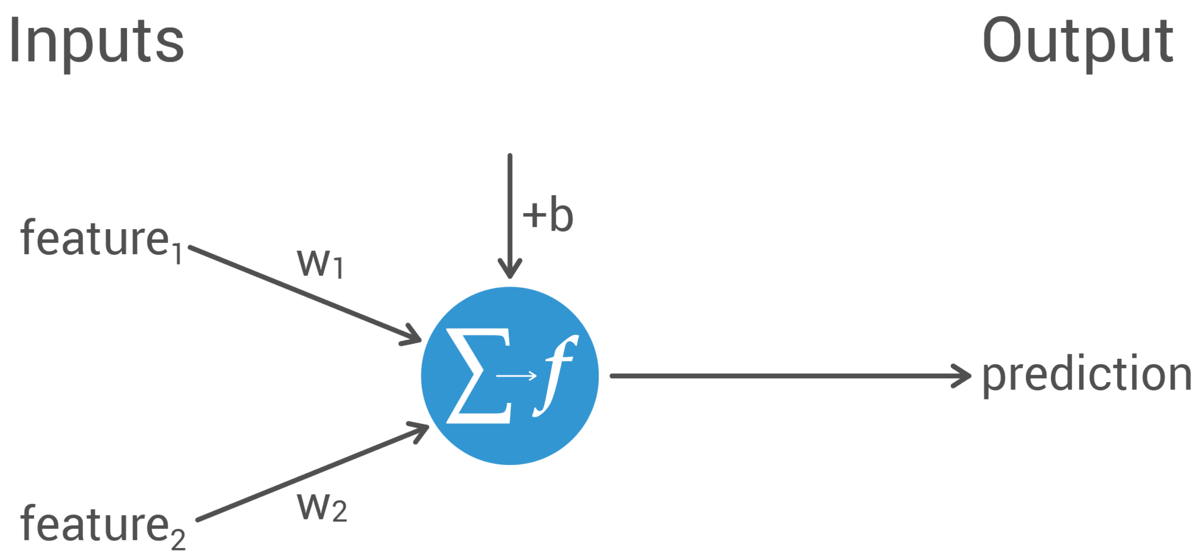
Diagram of a biological neuron

You can see the following:

- **The dendrites:** The branching structure of the dendrites receives signals as input from other neurons via synapses. These transmit the signal to the cell body where they are integrated into a signal.
- **The cell body & nucleus:** The integrated signal generates a nerve impulse at the beginning of the axon if the signal is strong enough and over a certain threshold.
- **Axon:** Here the nerve impulse is transmitted in one direction. it can cover large distances.
- **Synapses:** The signal is forwarded to other neurons here.

So in principle, you can say that neurons are objects that can pass on and filter signals. Only strong signals are passed on.

Now we can look at an artificial neuron. This is a mathematical model that imitates the basic functions of a neuron. The following image shows an artificial neuron.



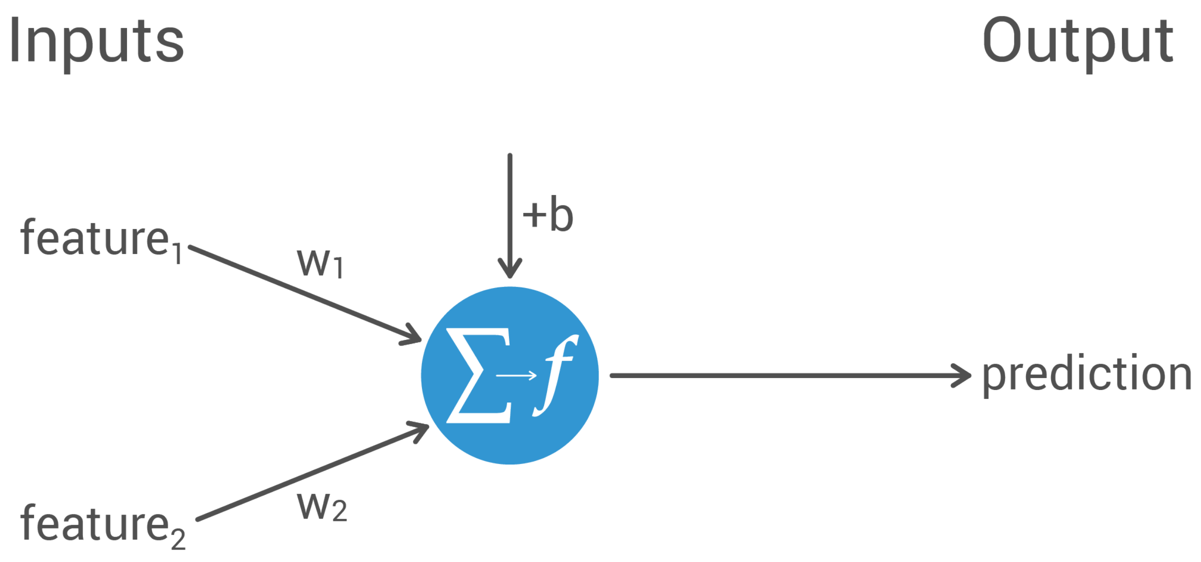


Diagram of a simple, one-layer neural network

We get this model from real neurons if we make the following abstractions:

- The **signal input** of the real neuron becomes the **features** of a data set. For the case above, you can imagine an example data set with two features, to keep things simple.
- The **dendrites** become the **weights** of the artificial neuron. Each feature, e.g. feature_1, corresponds to a weight, here w_1. This is equivalent to each dendrite carrying a part of the input signal in a real neuron. A single dendrite becomes a number **b** (without a feature or with a feature that always takes the value 1), the so-called ***bias*** of the artificial neuron.
- The output signal of the **synapses** becomes the model's predicted value. For example, this can be a probability prediction of whether a data point belongs to a class or not, or simply a number for a regression prediction. It's important to note here that: An artificial neuron only has one output.
- The **cell body**, **cell nucleus** and the **axon** are combined and put together in the bubble, with the characters $\Sigma \rightarrow f$. The symbols have the following meaning:

- First, all the features and weights (including the *bias*) are combined into a single number (just like a real neuron combines all the individual dendrite signals into a single signal). This is done using what's called a **linear combination**, which is defined by the symbol $\Sigma$. Each feature is multiplied by the corresponding weight and all the products are added together. In the diagram above this means: ## \begin{equation} \Sigma = \text{feature}_1 \cdot \text{w}_1 + \text{feature}_2\cdot \text{w}_2 + \text{b}~. \end{equation} The formula would then also become longer if there were more features (e.g. in our data set with 77 features we would have 78 terms in the sum).

- Next, $\Sigma$ is passed to the **activation function** ${f}$, which is represented by the arrow $\rightarrow$. This then directly calculates the value of the prediction. The activation function is therefore selected to reduce small values of $\Sigma$ very significantly. This corresponds to the property of real neurons that absorbs weak signals. What is missing now is the explicit form of the activation function. Depending on the problem, this can be selected very differently. We want to solve a binary classification problem. The *sigmoid* function is very often used as an activation function for this purpose: ## \begin{equation*} \text{prediction} = f_{\text{Sigmoid}} = \frac {1}{1 + e^{-\Sigma}} \end{equation*}

  Does this formula seem familiar? You already encountered it in *Chapter 2, Logistic Regression*. If we use this function as an activation function, then the artificial neuron carries out a logistic regression!

As a machine learning model, the artificial neuron will learn its weights $\text{w}$ based on the data. In supervised learning, the predictions are compared with true values and in unsupervised learning, the predictions are grouped by similarity. The artificial neuron, like all other machine learning models, uses the gradient method to determine the best weights here.

All in all we can draw the following conclusion:

Remember: A single artificial neuron is a generalization of logistic regression!

The artificial neuron is a generalization because the activation function can be selected in many ways, whereas in logistic regression it is always the sigmoid function.

Congratulations: You have learned what artificial neurons are and how they work. Now we want to concentrate on how to implement them practically.

Artificial neurons versus logistic regression

To get a first insight into the practical implementation of artificial neurons, we want to verify the claim that logistic regression is the same model as an artificial neuron with the *sigmoid* activation function. We'll do this by comparing the results from both models directly with each other and therefore simultaneously creating a first *baseline* model for our clients.

Since we have a lot of binary features, you should scale `features_train` and `features_val` first with the `MinMaxScaler` from `sklearn.preprocessing`.

Instantiate `MinMaxScaler()` as a variable named `scaler`. Then fit it to `features_train` and then transform `features_train` and `features_val`. Store the transformed values in the variables `features_train_scaled` and `features_val_scaled`.

You can ignore the resulting `DataConversionWarning` or use the following cell. Of course, it's also possible that you won't receive any warning at all. Some useful code that should be part of every data scientist's repertoire.

```
import warnings
from sklearn.exceptions import DataConversionWarning
warnings.filterwarnings(action='ignore', category=DataConversionWarning)
```

In [4]:
```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(features_train)
features_train_scaled = scaler.transform(features_train)
features_val_scaled = scaler.transform(features_val)

import warnings
from sklearn.exceptions import DataConversionWarning
warnings.filterwarnings(action='ignore', category=DataConversionWarning)
```

Perform the following steps to generate predictions using logistic regression:

- Import `LogisticRegression` from `sklearn.linear_model`
- Instantiate the model as `model_log`. Use the parameters `solver='lbfgs'`, `max_iter = 1000` and `C=1e42`.
- Fit `model_log` to `features_train_scaled` and `target_train`.
- Create predictions with `features_val_scaled`, store them as `target_val_pred_log` and return the result.

In [5]:
```
from sklearn.linear_model import LogisticRegression
model_log = LogisticRegression(solver='lbfgs', max_iter = 1000, C=1e42)
model_log.fit(features_train_scaled , target_train)
target_val_pred_log = model_log.predict(features_val_scaled)
```

Since we have a balanced class distribution in our data, we can use *accuracy* as a good evaluation metric. Now import `accuracy_score` from `sklearn.metrics` and print the accuracy achieved with the validation set.

In [7]:
```
from sklearn.metrics import accuracy_score
accuracy_score(target_val, target_val_pred_log)
```

Out[7]:
```
0.927075513870497
```

Congratulations: You've made a first prediction for the data set. With logistic regression you already achieved a very good accuracy of about 92.8% on the validation data!

The predictions with logistic regression were already quite good. We want to compare these with the artificial neuron. To do this, we'll use the module `tensorflow`. This is currently the most developed and widespread tool for implementing models with artificial neurons. Probably the biggest advantage of `tensorflow` is that it includes the `keras` module as an API. Thanks to `keras`, creating models with artificial neurons is very easy, because you do this in a way that is similar to `sklearn`. In just a few lines of code, the following code cell contains everything you need for a model with one artificial neuron.

Attention: The different versions of `tensorflow` sometimes have a considerable influence on how you use it. `keras` is only integrated from `tensorflow` 2.x , which made `tensorflow` extremely powerful and simple. In this course we use `tensorflow` 2.1.0. The code is compatible with all versions that have a 2 at the beginning. The code can't be **directly migrated** to all versions with a 1 in front (but there are ways to make them compatible, see here). We (and also the developers of `keras` and `tensorflow`) recommend using `tensorflow` 2.x.

Run the cell (ignore warnings of there are any).

In [8]:
```
from tensorflow.ktarget_val_pred_logort Sequential
from tensorflow.keras.layers import Dense

model_an = Sequential()  # define the model type
model_an.add(Dense(1, activation='sigmoid', input_dim=features_train_scaled.shape[1]))
model_an.compile(optimizer = "adam", loss = 'binary_crossentropy', metrics = ['accurac
```

```
2024-04-26 18:35:38.653807: W tensorflow/stream_executor/platform/default/dso_loader.
cc:60] Could not load dynamic library 'libcudart.so.11.0'; dlerror: libcudart.so.11.
0: cannot open shared object file: No such file or directory
2024-04-26 18:35:38.653846: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Igno
re above cudart dlerror if you do not have a GPU set up on your machine.
2024-04-26 18:35:41.175393: I tensorflow/compiler/jit/xla_cpu_device.cc:41] Not creat
ing XLA devices, tf_xla_enable_xla_devices not set
2024-04-26 18:35:41.175562: W tensorflow/stream_executor/platform/default/dso_loader.
cc:60] Could not load dynamic library 'libcuda.so.1'; dlerror: libcuda.so.1: cannot o
pen shared object file: No such file or directory
2024-04-26 18:35:41.175574: W tensorflow/stream_executor/cuda/cuda_driver.cc:326] fai
led call to cuInit: UNKNOWN ERROR (303)
2024-04-26 18:35:41.175597: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:15
6] kernel driver does not appear to be running on this host (7c20761d1d14): /proc/dri
ver/nvidia/version does not exist
2024-04-26 18:35:41.175939: I tensorflow/core/platform/cpu_feature_guard.cc:142] This
TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to us
e the following CPU instructions in performance-critical operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler
flags.
2024-04-26 18:35:41.179434: I tensorflow/compiler/jit/xla_gpu_device.cc:99] Not creat
ing XLA devices, tf_xla_enable_xla_devices not set
```

In just 3 lines of code, we have defined an artificial neuron, under the name `model_an`. To train it, we can use the `my_model.fit()` method, just like in `sklearn`. The training then takes place in what we call epochs, and this can take a little while. Run the next code cell to familiarize yourself with the output. In the next lesson, we'll go into more detail and clarify all the terms we are currently using.

In [9]:
```
model_an.fit(features_train_scaled, target_train, epochs=5) # Fit the model
```

```
2024-04-26 18:38:32.599614: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.c
c:116] None of the MLIR optimization passes are enabled (registered 2)
2024-04-26 18:38:32.616554: I tensorflow/core/platform/profile_utils/cpu_utils.cc:11
2] CPU Frequency: 2495290000 Hz
Epoch 1/5
8654/8654 [==============================] - 11s 1ms/step - loss: 0.2915 - accuracy:
0.8981
Epoch 2/5
8654/8654 [==============================] - 10s 1ms/step - loss: 0.1372 - accuracy:
0.9274
Epoch 3/5
8654/8654 [==============================] - 11s 1ms/step - loss: 0.1351 - accuracy:
0.9272
Epoch 4/5
8654/8654 [==============================] - 10s 1ms/step - loss: 0.1340 - accuracy:
0.9282
Epoch 5/5
8654/8654 [==============================] - 11s 1ms/step - loss: 0.1352 - accuracy:
0.9274
```

Out[9]:
```
<tensorflow.python.keras.callbacks.History at 0x7fc678184880>
```

Now let's compare our artificial neuron with the logistic regression. To do this, we need the predictions `model_an` makes based on the validation data. We can use a method we're familiar with to do this, just like for the training. Pass `features_val_scaled` to `model_an.predict()` and store the result as `target_val_pred_an` and print it.

In [11]:
```
target_val_pred_an = model_an.predict(features_val_scaled)
```

In contrast to `LogisticRegression`, the `keras` model outputs a nested array with the results of the activation function. First we'll convert this into a 1-dimensional array. We can achieve this with the `my_array.flatten()` method.

Use this method and store the result as `target_val_pred_an` again. Print the result.

In [12]:
```
target_val_pred_an = target_val_pred_an.flatten()
```

To calculate the accuracy, we now have to convert the predictions to the discrete values `0` and `1`. We'll use `0.5` as the threshold value, just like with `LogisticRegression`. So replace all values smaller than `0.5` in `target_val_pred_an` with `0` and all the other values with `1`. Print the result.

Tip: The values `False` and `True` are interpreted by `Accuracy_score` as `0` and `1`.

In [18]:
```
target_val_pred_an = target_val_pred_an > 0.5
#target_val_pred_an = [1 if True else 0 for value in target_val_pred_an]
```

What accuracy do we get for our artificial neural network?

```
In [19]: accuracy_score(target_val, target_val_pred_an)
```

```
Out[19]: 0.9272610723593762
```

`tensorflow` also offers us a shortcut for the steps above. Instead of first generating the predictions, then converting the values into discrete values and finally calculating the *accuracy*, we can call the `my_model.evaluate()` method. This takes the validation data as input and outputs two values. Run the following code cell to see the result.

```
In [20]: model_an.evaluate(features_val_scaled, target_val) # Evaluate the model
```

```
3706/3706 [==============================] - 3s 738us/step - loss: 0.1353 - accuracy:
0.9273
Out[20]: [0.13533106446266174, 0.9272610545158386]
```

There are two values. The first value belongs to the `loss` function, which makes quantitative statements about how well our model has learned mathematically. The second value is the *accuracy* or metric we set in `my_model.compile()`. In the next lesson, we'll go into this in more detail.

All in all, we see: The artificial neuron also achieves an almost identical *accuracy score* of about 92.8% with the *sigmoid* activation function (slight deviations are always possible due to statistical fluctuations). So we can see that our assumption that an artificial neuron with *sigmoid* activation function is identical to logistic regression, is correct, both theoretically and mathematically. Nevertheless, the most critical difference (which you have probably noticed) is how long it take to train both models! A typical characteristic of artificial neurons is that it takes much longer to train them. Generalization has its price!

Congratulations: You've trained and used your first artificial neuron! You now know that an artificial neuron gives *input* values a weight and transforms them with an activation function. Your superiors at *Lemming Loans Limited* are pleased that you have already created a first AN. In the next lesson we'll use the human brain as a model to connect artificial neurons to form artificial neural networks and unleash their true power.

Remember:

- Artificial neurons can be implemented quickly with `tensorflow` in a similar way to `sklearn`.
- An artificial neuron assigns *weights* and a *bias* to the *features* and transforms the data with an activation function.
- An artificial neuron is a generalization of logistic regression!

Do you have any questions about this exercise? Look in the forum to see if they have already been discussed.