Introduction to Artificial Neural Networks - Neural Networks

Module 2 | Chapter 5 | Notebook 4

In the last exercise we looked at an artificial neuron and learned how to implement it using `tensorflow`. Now we want to increase the complexity of the model and combine several artificial neurons into one model, an artificial neural network, ANN for short. By the end of this exercise you will have done the following:

- Learned what ANNs are and how they work.
- Understand the terms *epoch* and *batch size*.
- Generated an ANN yourself, using `tensorflow`.

Artificial neural networks

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.

Now let's import our data quickly by running the following code cell:

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()
```

Out[1]:

	grade_A	grade_B	grade_C	grade_D	grade_E	grade_F	grade_G	sub_grade_A1	sub_grade_A2	sul
0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	
1	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	
2	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	
3	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
4	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	

5 rows × 77 columns

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

Column number	Column name	Type	Description
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*

Since we have a lot of binary categories, we should scale the data with the `MinMaxScaler`. Execute the following code cell to do this and ignore the `DataConversionWarning` if you receive it:

In [2]:
```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
features_train_scaled = scaler.fit_transform(features_train)
features_val_scaled = scaler.transform(features_val)
```

In the last lesson, you learned that artificial neurons are modelled on real, biological neurons in the human brain. But there is one question we haven't clarified. Why is it even interesting to use our brain as a model for machine learning models? Fascinatingly, a lot of scientists from a wide range of disciplines agree that the human brain is the most complicated system in the universe that we know about. But apart from that, our brains are by far the best supercomputers in the world and we haven't yet established what the upper limits are. So it makes sense to take the human brain as a model. If you're interested in this, you should have a look at this link and have a look around).

It's true that neurons are essential components of brain performance, but individual neurons are not so useful by themselves. The true strength of neurons is revealed through their connections with each other in neural networks. In the brain, these networks are highly complex and for the most part not understood. Nevertheless, the principle of connecting neurons can be directly applied to artificial neurons.

As you can imagine, there are many different ways to connect artificial neurons. We call this the **architecture of neural networks**. The neural networks we build in this exercise are what we call feedforward neural networks. This means that the information in the network only flows in one direction, i.e. forward. What does that mean?

The artificial neurons are first arranged side by side. We call this a layer. The neurons within a layer are not connected to one another. If we apply what we learned in the last lesson, a layer is nothing more than an ensemble (see *Module 2, Chapter 3*) of generalized logistic regressions, independent models that produce their own results.

An important part of the ensembles is that the independent models are brought together and produce a single result. Depending on the result you want, you have several options here. With a binary classification problem (like this one), you merge the results from all the artificial neurons in the layer into a new layer with a single artificial neuron. This new layer is called the

output layer, the layer that produces the final result. This may remind you of the concept of *stacking* machine learning models - (reference link). This kind of comparison is good to gain an initial understanding, but there are still subtle differences in their mathematical implementation (we go into these, since this would go far beyond the scope of our course).

Now you can also take the results from a layer, to another layer, just like stacking. This means the ANN is drawn out and the layers are connected. We distinguish between two types:

- **input layer**: The first layer in an ANN is formed from the data set used. Each feature is assigned to a single artificial neuron. So this layer always has as many artificial neurons as there are features in the data set. This layer does not use any bias.
- **Hidden layer**: All layer between the input and output layers.

Now let's look at an example in the following image:

ANN with hidden layer

"body weight" and "height" here are example features representing single artificial neurons which are passed to the ANN. The hidden layer contains 2 artificial neurons. Each neuron applies its own weights and bias to the inputs. The results of the hidden layer are passed to the output layer, which produces our predictions about the hypothetical class "gender".

This is a feedforward ANN. Typically, single layers contain hundreds to thousands of artificial neurons. So the total number of weights that have to be learned, increases very quickly. This often results in ANNs taking a very long time to train.

Congratulations: Now you know what a feedforward ANN is and how it works.

Artificial neural networks in `tensorflow`

The `tensorflow` module offers a simple procedure in its submodule `tensorflow.keras` for building a feedforward ANN. The model is called `Sequential` and it can be found in `tensorflow.keras.models`. Just like in `sklearn`, `Sequential` is instantiated and assigned to a variable.

The layers are located in `tensorflow.keras.layers`. A layer where the artificial neurons are arranged side by side without influencing each other, is called a `dense` layer. There are a lot more layer types, but `Dense` is perfectly adequate for binary classification problems.

Let's now begin building a feedforward ANN. Now import `Sequential` from `tensorflow.keras.models` and `Dense` from `tensorflow.keras.layers`.

In [3]:
```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

```
2024-04-28 20:35:32.014656: 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
2024-04-28 20:35:32.014693: 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.
```

Instantiate `Sequential()` as the variable `model_ann`.

In [4]:
```
model_ann =Sequential()
```

```
2024-04-28 20:35:55.294351: I tensorflow/compiler/jit/xla_cpu_device.cc:41] Not creat
ing XLA devices, tf_xla_enable_xla_devices not set
2024-04-28 20:35:55.294565: 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
2024-04-28 20:35:55.294578: W tensorflow/stream_executor/cuda/cuda_driver.cc:326] fai
led call to cuInit: UNKNOWN ERROR (303)
2024-04-28 20:35:55.294597: 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
2024-04-28 20:35:55.294769: 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.
2024-04-28 20:35:55.298110: I tensorflow/compiler/jit/xla_gpu_device.cc:99] Not creat
ing XLA devices, tf_xla_enable_xla_devices not set
```

The model is now ready to receive and understand layers. First let's define the layers we want to have in the ANN. We want a *hidden layer* with `35` artificial neurons and an output layer with an artificial neuron. For each layer we'll instantiate `Dense` in a variable and with the appropriate hyperparameters - (link to documentation). The most important hyperparameters are:

```
Dense(units = int,         # The dimensionality of the output space
      activation = str,       # The activation function
      input_dim = int/tuple # Number of dimensions of the features
     )
```

First we have `units`, which determine the number of artificial neurons in the layer. Next we have `activation`, which defines the activation function. There is a large selection of built-in activation functions - (link to the documentation). We'll limit ourselves to the following two:

- `activation = 'sigmoid'`: The Sigmoid function, from logistic regression. For binary classification problems, this is the first choice for the output layer.
- `activation = 'relu'`: The *Rectified Linear Unit* function, ReLU for short, is a very good choice for hidden layers. It calculates the Σ value (see previous lesson) of the artificial neuron and the *output* becomes either Σ if Σ is greater than 0, or 0 if Σ is less than 0. It's a very simple function.

The last argument is `input_dim`. This is only relevant to us in the first layer, the hidden layer with 35 artificial neurons. This argument creates an input layer before the first layer. For this it's necessary to know how many dimensions the data set has. In our case, it's the number of features. In more complicated cases, these can be images with several color channels, which have features along several axes, called tensors (hence the name `tensorflow`).

Now it's your turn again. Create two instances of `Dense`.

Store the first one in the variable `hidden_layer`. In this layer, we want 35 artificial neurons, the `'relu'` activation function and, since it is the first layer, an `input_dim` matching the number of features in `features_train`.

- You store the second one under `output_layer`. We only want a single artificial neuron with a sigmoid activation function here.

Tip: You can get the number of features by using the `my_df.shape` attribute from `features_train`, for example.

In [10]:
```
hidden_layer = Dense(units = 35, activation = 'relu', input_dim = features_train.shape
output_layer = Dense(units = 1, activation = 'sigmoid')
```

Now we'll add the layers to the `Sequential` model. You can use the `my_model.add()` method to do this. You pass each layer **individually** to the model by specifying it as an argument in the `.add()` method. For example, the syntax for a three layer ANN would be:

```
my_model.add(layer_1)           # Adding first hidden layer
my_model.add(layer_2)           # Adding second hidden layer
my_model.add(layer_3)           # Adding output layer
```

The order is also important here. Now add our generated layers to our model, in the correct order.

In [11]:
```
model_ann.add(hidden_layer)
model_ann.add(output_layer)
```

Now the ANN is finished. But to be able to do calculations with it, we have to convert it into machine code, i.e. compile it. This is the task of the `my_model.compile()` method. The following parameters are specified:

```
my_model.compile(optimizer = str,
              loss = str,
              metrics = [list])
```

Like many other models from `sklearn`, you can add an optimizer to the neural network. This is the algorithm that gradually removes the `loss` in the background: see *Logistic Regression (Module 2 Chapter 2)*. In the neural network, the weights of the individual neurons are modified in such a way that the predictions for the features in the training data always match the actual target vector better. How well they fit is evaluated by the `loss`.

`tensorflow.keras` offers us some cost functions and optimizers. For binary classification using probabilities, binary cross entropy is usually used (`loss='binary_crossentropy'`).

The optimizer changes the weights and tries to reduce the `loss` in the process. `optimizer='adam'` is often used because it produces robust results with relatively low memory requirements.

The metric you specify is used to evaluate how good the model is. So it is not optimized directly (like the cost function), but indirectly. `tensorflow.keras` allows us to specify several metrics, so a `list` or a `dict` is expected. Here you can find a list of possible metrics provided by `tensorflow.keras`. Use `metric=['accuracy']` here.

Now compile the model. You don't have to assign it to a variable.

```
In [14]: model_ann.compile(optimizer='adam',loss='binary_crossentropy', metrics=['accuracy'])
```

As you saw in the last exercise, fitting the model is very similar to in `sklearn`. You pass the features and the target vector of the training set to the `my_model.fit() method`. However, you specify two more parameters, `epochs`, and `batch_size`.

epochs indicates how often the training process iterates through all the training data. It can be useful for the model to see the training data several times. This is because the model optimizes the weights step by step. It's possible that not enough steps are taken to achieve a good result. Then you train the model further with more epochs. However, too many epochs can also lead to overfitting.

Usually, not all the data points are handed over at once during training. This is controlled with the **batch size**. The `batch_size` parameter determines how many data points the model uses to adjust the weights. For example, if you have 10 data points in the training set and specify a `batch_size` of 2, each epoch will consist of 5 optimization steps. These steps are called iterations.

The procedure of only using the data bit by bit for training has the advantage that it requires much less memory. Especially with large amounts of data, the main memory is otherwise not sufficient for the training. In addition, the weights of the neural network are updated with each batch and not just once the model has gone through all of the training data. So there are more optimization steps, which results in the model training more quickly. However, this also has the disadvantage that the adjustment of the weights is less accurate. In practice, 32 to 512 data points are usually used as the `batch_size` to achieve good results.

Fit the model to the scaled training data in `features_train_scaled`. Use the parameters `epochs=5` and `batch_size=64`.

```
In [15]: model_ann.fit(features_train_scaled, target_train, epochs=5, batch_size=64 )
```

```
2024-04-28 21:21:46.722205: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.c
c:116] None of the MLIR optimization passes are enabled (registered 2)
2024-04-28 21:21:46.744532: I tensorflow/core/platform/profile_utils/cpu_utils.cc:11
2] CPU Frequency: 2495290000 Hz
```

```
Epoch 1/5
4327/4327 [==============================] - 6s 1ms/step - loss: 0.1928 - accuracy:
0.9095
Epoch 2/5
4327/4327 [==============================] - 5s 1ms/step - loss: 0.1348 - accuracy:
0.9277
Epoch 3/5
4327/4327 [==============================] - 5s 1ms/step - loss: 0.1342 - accuracy:
0.9283
Epoch 4/5
4327/4327 [==============================] - 5s 1ms/step - loss: 0.1335 - accuracy:
0.9292
Epoch 5/5
4327/4327 [==============================] - 5s 1ms/step - loss: 0.1331 - accuracy:
0.9291
```

Out[15]:

```
<tensorflow.python.keras.callbacks.History at 0x7f98d00443d0>
```

Now we just need to evaluate the model. You can use the `my_model.evaluate()` method to do this -(link to documentation). It needs two arguments. The first is the validation data of the features. The second is the target vector of the validation data. The syntax is as follows:

```
my_model.evaluate(my_feature_val,    # Validation feature data
                  my_target_val)     # Validation target vector
```

It outputs a vector. Its first entry is the value of the loss function, and its second value is the value of the metric specified in the `.compile()` method, so in this case the accuracy. If a list of metrics has been passed, more values are output. The order is the same as when compiling.

Now evaluate the model and store the result in the variable `model_ann_eval`. Print the variable.

In [16]:

```
model_ann_eval = model_ann.evaluate(features_val_scaled, target_val)
model_ann_eval
```

```
3706/3706 [==============================] - 2s 631us/step - loss: 0.1327 - accuracy:
0.9294
```

Out[16]:

```
[0.13273273408412933, 0.9294118881225586]
```

It should achieve an accuracy of about 92.9%. So a slight improvement to the single artificial neuron of the last lesson. It doesn't sound like much, but it can be worth a lot depending on the circumstances!

If you like, you can now go back to the model and play around with the hyperparameters. See what effects various optimizers and loss functions have on the final result.

Congratulations: Now you know how to build a neural network with `tensorflow`, fit it to the data and evaluate it. Next, we'll look at the overfitting problem!

Remember:

- Instantiate the neural network with `Sequential()` and the corresponding layers with `Dense()`.

- Add each layer one after the other with `my_model.add()`. Pay attention to the order the steps come in.
- The last layer generates the predictions, so you should specify the following parameters `units=1` and `activation='sigmoid'`.
