

Regularizing Artificial Neural Networks

Module 2 | Chapter 5 | Notebook 5

In the last exercise you used `tensorflow` itself to define a first artificial neural network and apply it to the data. Now we'll try to improve the ANN. To do this, we'll incorporate more hidden layers and regularize the ANN. By the end of this lesson you will know:

- How to identify overfitting in ANNs.
 - What a *dropout layer* is.
 - What the term *early stopping* means.
-

Deep 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.

Our two-layer artificial neural network from the last lesson has already achieved a very good accuracy of 92.9%. Now let's see if we can improve the result by adding more layers.

Run the following code cell to import the data and scale it immediately with `MinMaxScaler`.

```
In [1]: # read data
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')

# scale data
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
features_train_scaled = scaler.fit_transform(features_train)
features_val_scaled = scaler.transform(features_val)
```

```
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	sub_grade_A3
0	0.0	0.0	0.0	1.0	0.0	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	0.0
2	0.0	0.0	1.0	0.0	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	0.0
4	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

5 rows × 11 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

Column number	Column name	Type	Description
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 <i>Presenting and Preparing Loan Data</i>

Artificial neural networks are multilayer neural networks if they have at least one hidden layer. If they have one hidden layer, we also refer to them as shallow neural networks. Theoretically, one hidden layer is already enough to create very complex functions. But in practice, a neural network is easier to train if it has several hidden layers. Then we call this a deep neural network and the term **deep learning**.

Before we build our first deep learning model, we need to instantiate it. Now instantiate a feedforward ANN (`Sequential`), using `tensorflow.keras` , and give it the variable name `model_ann` .

```
In [2]: from tensorflow.keras import Sequential
model_ann = Sequential()
```

```
2024-04-29 14:21:08.297274: 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
2024-04-29 14:21:08.297310: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if you do not have a GPU set up on your machine.
2024-04-29 14:21:09.472357: I tensorflow/compiler/jit/xla_cpu_device.cc:41] Not creating XLA devices, tf_xla_enable_xla_devices not set
2024-04-29 14:21:09.472510: W tensorflow/stream_executor/platform/default/dso_loader.cc:60] Could not load dynamic library 'libcuda.so.1'; dlerror: libcuda.so.1: cannot open shared object file: No such file or directory
2024-04-29 14:21:09.472525: W tensorflow/stream_executor/cuda/cuda_driver.cc:326] failed call to cuInit: UNKNOWN ERROR (303)
2024-04-29 14:21:09.472545: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:156] kernel driver does not appear to be running on this host (7c20761d1d14): /proc/driver/nvidia/version does not exist
2024-04-29 14:21:09.472739: I tensorflow/core/platform/cpu_feature_guard.cc:142] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations: AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2024-04-29 14:21:09.476645: I tensorflow/compiler/jit/xla_gpu_device.cc:99] Not creating XLA devices, tf_xla_enable_xla_devices not set
```

We can assign different [activation functions](#) to the hidden layers. The functions are non-linear as this gives the ANN much more flexibility to separate the data. Other questions that arise in relation to the hidden layers are those about the number of layers and the number of their artificial neurons (units). There are no set rules here. One approach is to start with just one or two hidden layers and add more and more until you end up overfitting. A similar approach can

be taken with the number of neurons, usually using values between the number of input features and output classes. The number of neurons is often reduced from layer to layer.

Another approach is to start straight away with a relatively large number of layers and neurons and to end the training more quickly or regularize the network if you encounter overfitting. You can also use the same number of neurons for each *hidden layer*. This way you effectively have only one hyperparameter (the number of neurons) instead of one for each layer. We'll try out this approach now.

Add 5 `Dense` layers of 50 neurons each to your model. Use `'relu'` as the activation function. Remember to give the first layer (and only the first layer) the dimensions of the input.

Tip: Unfortunately, you **can't** just define two layers, the first one under the name `hidden_first` and the second one under `hidden_second`, and then simply add the second layer to the model four times. You really have to define each layer individually, including `hidden_third`, `hidden_fourth` and `hidden_fifth`.

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

hidden_first = Dense(units = 5, activation = 'relu', input_dim = features_train.shape[1])
hidden_second = Dense(units = 5, activation = 'relu')
hidden_third = Dense(units = 5, activation = 'relu')
hidden_fourth = Dense(units = 5, activation = 'relu')
hidden_fifth = Dense(units = 5, activation = 'relu')

model_ann.add(hidden_first)
model_ann.add(hidden_second)
model_ann.add(hidden_third)
model_ann.add(hidden_fourth)
model_ann.add(hidden_fifth)
```

Finally, we need the output layer. As before, this only needs one unit and the `'sigmoid'` activation function. Add the appropriate output layer, under the name `output_layer`, to the model.

```
In [4]: output_layer = Dense(units = 1, activation = 'sigmoid')

model_ann.add(output_layer)
```

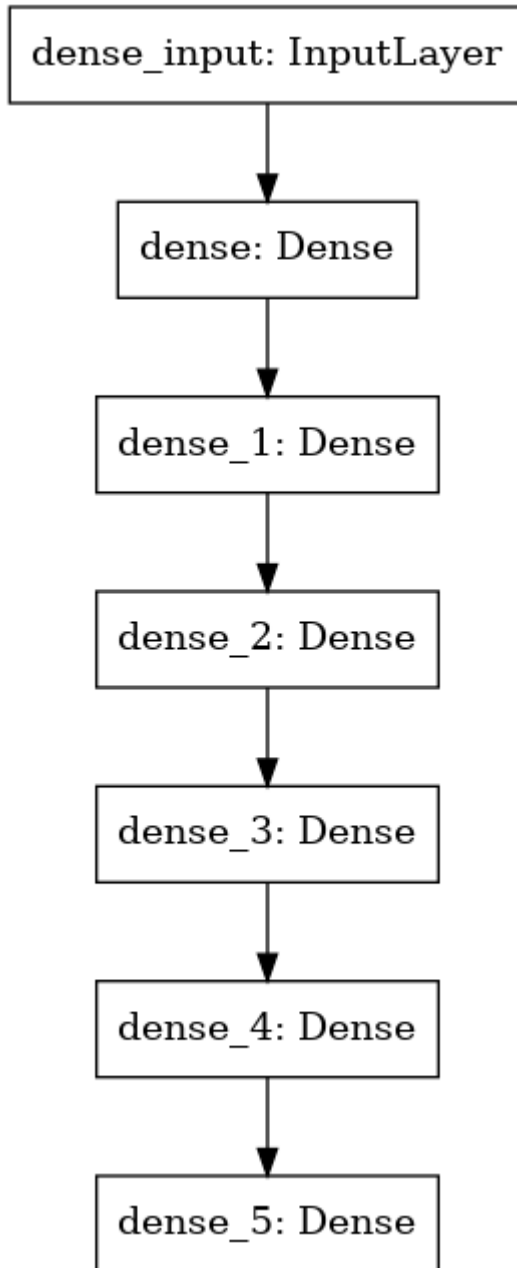
Now compile `model_ann`. Use the parameters `optimizer="adam"`, `loss='binary_crossentropy'`, `metrics=['accuracy']`, just like in the last exercise.

```
In [5]: model_ann.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

After compiling we can display our model visually. For this we need the function `plot_model()` from `tensorflow.keras.utils`. This can be extremely useful if you have a lot of layers and want to check if everything was created correctly. Run the following cell to display them to get an insight into the architecture of our model.

```
In [6]: # Import plot_model
from tensorflow.keras.utils import plot_model
# plot model
plot_model(model_ann)
```

Out[6]:



Now it's time to train the model. So far we have adapted the ANN to all the training data and evaluated the model afterwards with validation data. Conveniently, `tensorflow.keras` offers us the possibility to enter the validation data when training, so that we can look at the metrics based on it directly during training. For this purpose, we pass the parameter `validation_data` with a tuple of features and target vector of the validation data to the `my_model.fit()` method. In our case `validation_data=(features_val_scaled, target_val)`. The metrics and loss values for training and validation data can also be stored for each epoch. To do this, the `my_model.fit()` method generates a `History` object. This can help decide how many epochs are actually needed.

Deep Dive: A form of **backpropagation** (backprop or BP for short) is always used when training ANNs. This algorithm works as follows:

1. The model makes a prediction for each data point in the current training step. Then the mean loss of these data points is determined.
2. The contribution of each weight in the last layer to the loss is calculated.
3. For each additional layer, all weights' contributions are calculated (from behind).
4. The weights are adjusted according to their contributions.
5. The steps are repeated.

The calculation of the contributions has to be carried out from the last to the first layer, hence the name of this procedure.

Fit `model_log` to `features_train_scaled` and `target_train`. Also pass the validation data. Use 20 epochs and a `batch_size` of 64. Store the result as a variable named `hist_ann`. It may take a few minutes to run the cell.

```
In [7]: hist_ann = model_ann.fit(features_train_scaled,
                                target_train,
                                epochs=20,
                                batch_size=64,
                                validation_data=(features_val_scaled, target_val))
```

```
2024-04-29 14:21:09.739143: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.c
c:116] None of the MLIR optimization passes are enabled (registered 2)
2024-04-29 14:21:09.760534: I tensorflow/core/platform/profile_utils/cpu_utils.cc:11
2] CPU Frequency: 2495290000 Hz
```

Epoch 1/20
4327/4327 [=====] - 8s 2ms/step - loss: 0.2269 - accuracy: 0.9000 - val_loss: 0.1363 - val_accuracy: 0.9259

Epoch 2/20
4327/4327 [=====] - 7s 2ms/step - loss: 0.1362 - accuracy: 0.9271 - val_loss: 0.1354 - val_accuracy: 0.9270

Epoch 3/20
4327/4327 [=====] - 7s 2ms/step - loss: 0.1356 - accuracy: 0.9275 - val_loss: 0.1351 - val_accuracy: 0.9267

Epoch 4/20
4327/4327 [=====] - 7s 2ms/step - loss: 0.1350 - accuracy: 0.9273 - val_loss: 0.1358 - val_accuracy: 0.9270

Epoch 5/20
4327/4327 [=====] - 7s 2ms/step - loss: 0.1342 - accuracy: 0.9275 - val_loss: 0.1343 - val_accuracy: 0.9271

Epoch 6/20
4327/4327 [=====] - 7s 2ms/step - loss: 0.1341 - accuracy: 0.9276 - val_loss: 0.1344 - val_accuracy: 0.9269

Epoch 7/20
4327/4327 [=====] - 7s 2ms/step - loss: 0.1344 - accuracy: 0.9284 - val_loss: 0.1349 - val_accuracy: 0.9278

Epoch 8/20
4327/4327 [=====] - 7s 2ms/step - loss: 0.1326 - accuracy: 0.9294 - val_loss: 0.1329 - val_accuracy: 0.9288

Epoch 9/20
4327/4327 [=====] - 7s 2ms/step - loss: 0.1325 - accuracy: 0.9298 - val_loss: 0.1325 - val_accuracy: 0.9285

Epoch 10/20
4327/4327 [=====] - 7s 2ms/step - loss: 0.1320 - accuracy: 0.9297 - val_loss: 0.1323 - val_accuracy: 0.9291

Epoch 11/20
4327/4327 [=====] - 7s 2ms/step - loss: 0.1312 - accuracy: 0.9303 - val_loss: 0.1323 - val_accuracy: 0.9293

Epoch 12/20
4327/4327 [=====] - 7s 2ms/step - loss: 0.1316 - accuracy: 0.9295 - val_loss: 0.1322 - val_accuracy: 0.9290

Epoch 13/20
4327/4327 [=====] - 7s 2ms/step - loss: 0.1319 - accuracy: 0.9292 - val_loss: 0.1321 - val_accuracy: 0.9294

Epoch 14/20
4327/4327 [=====] - 7s 2ms/step - loss: 0.1317 - accuracy: 0.9297 - val_loss: 0.1327 - val_accuracy: 0.9299

Epoch 15/20
4327/4327 [=====] - 7s 2ms/step - loss: 0.1312 - accuracy: 0.9305 - val_loss: 0.1321 - val_accuracy: 0.9291

Epoch 16/20
4327/4327 [=====] - 7s 2ms/step - loss: 0.1310 - accuracy: 0.9307 - val_loss: 0.1319 - val_accuracy: 0.9291

Epoch 17/20
4327/4327 [=====] - 7s 2ms/step - loss: 0.1312 - accuracy: 0.9299 - val_loss: 0.1322 - val_accuracy: 0.9291

Epoch 18/20
4327/4327 [=====] - 7s 2ms/step - loss: 0.1304 - accuracy: 0.9303 - val_loss: 0.1322 - val_accuracy: 0.9294

Epoch 19/20
4327/4327 [=====] - 7s 2ms/step - loss: 0.1322 - accuracy: 0.9297 - val_loss: 0.1324 - val_accuracy: 0.9292

Epoch 20/20
4327/4327 [=====] - 7s 2ms/step - loss: 0.1320 - accuracy: 0.9298 - val_loss: 0.1319 - val_accuracy: 0.9299

With `features_val_scaled`, we achieve an accuracy of about 93.0%. This is already slightly above the ANN's prediction from the last exercise. Now we can use the `History` object to view the model's training curve. It has an attribute called `my_hist.history`. This is a `dict` containing both the loss values and our metrics after each training period. The relevant keys are called `'loss'` and `'accuracy'` for the training data, and `'val_loss'` and `'val_accuracy'` for the validation data.

Run the following code cell to get an insight into the `History` object.

```
In [8]: # Print the model history
print(hist_ann.history)

{'loss': [0.15975850820541382, 0.13621434569358826, 0.13568446040153503, 0.1352080702
7816772, 0.13491719961166382, 0.1340339630842209, 0.13324840366840363, 0.132814064621
92535, 0.13256219029426575, 0.13235832750797272, 0.1321592777967453, 0.13201676309108
734, 0.13199806213378906, 0.13197630643844604, 0.13189052045345306, 0.131809756159782
4, 0.13176363706588745, 0.13175177574157715, 0.1317688524723053, 0.1317437291145324
7], 'accuracy': [0.9210366606712341, 0.926854133605957, 0.9271177053451538, 0.9269696
474075317, 0.9271068572998047, 0.9277604818344116, 0.9288907647132874, 0.929157972335
8154, 0.92940354347229, 0.9291651844978333, 0.9297176599502563, 0.9295660257339478,
0.9295299053192139, 0.9295588135719299, 0.9298657178878784, 0.9296743273735046, 0.929
6382665634155, 0.9296707510948181, 0.9298007488250732, 0.9298476576805115], 'val_los
s': [0.13634103536605835, 0.13536496460437775, 0.13512642681598663, 0.135761380195617
68, 0.13432104885578156, 0.13442900776863098, 0.1348770707845688, 0.1328797489404678
3, 0.13252148032188416, 0.13231469690799713, 0.1322726160287857, 0.13221505284309387,
0.13206838071346283, 0.1326867640018463, 0.13207019865512848, 0.13190115988254547, 0.
13215990364551544, 0.13220086693763733, 0.13243117928504944, 0.13188773393630981], 'v
al_accuracy': [0.9259452819824219, 0.9270249009132385, 0.9267212748527527, 0.92695742
84553528, 0.9271177053451538, 0.9268814921379089, 0.9278177618980408, 0.9288383126258
85, 0.9284924864768982, 0.9290913343429565, 0.9293444156646729, 0.9290154576301575,
0.9294202923774719, 0.9299347996711731, 0.9290829300880432, 0.9291250705718994, 0.929
1419386863708, 0.929386556148529, 0.9292347431182861, 0.9299179315567017]}
```

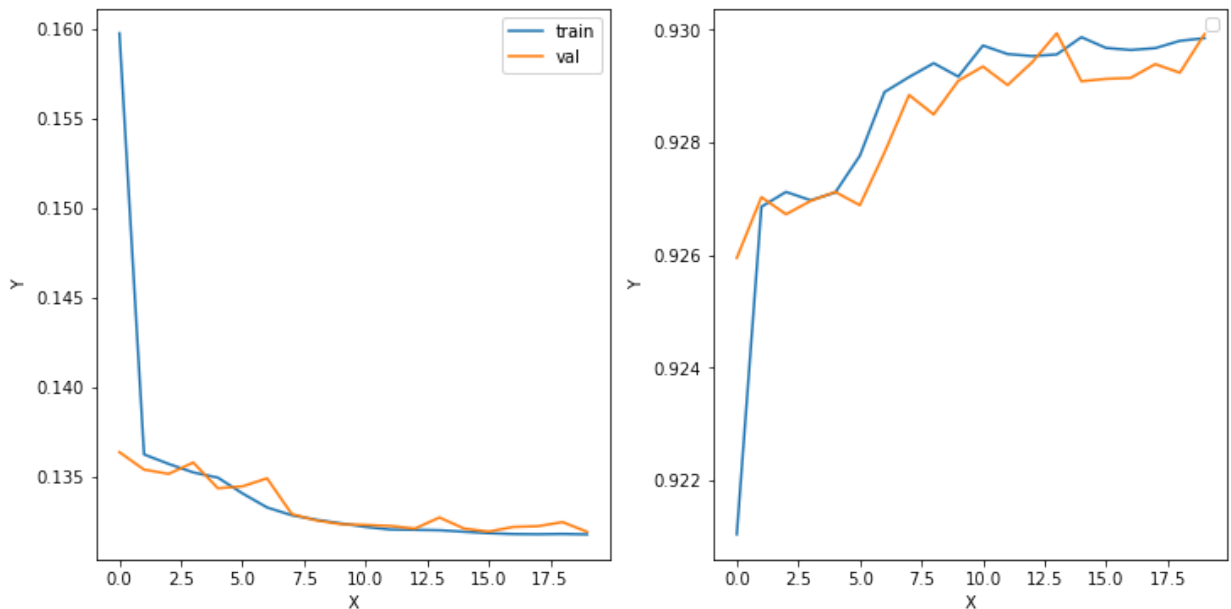
Now create 2 line graphs which compare the values of the training and validation data when adjusting the model. The first should show the loss values and the second the accuracy values.

```
In [14]: import matplotlib.pyplot as plt
fig, axs = plt.subplots(ncols=2, figsize=(12,6))
#, hist_ann.history['val_accuracy']
axs[0].plot(range(0,20),hist_ann.history['loss'], label='train')
axs[0].plot(range( 0,20),hist_ann.history['val_loss'], label='val')
axs[1].plot(range( 0,20),hist_ann.history['accuracy'] )
axs[1].plot( range( 0,20), hist_ann.history['val_accuracy'])


axs[0].set(xlabel='X', ylabel='Y')
axs[0].legend()
axs[1].set(xlabel='X', ylabel='Y')
axs[1].legend()
```

No handles with labels found to put in legend.

```
Out[14]: <matplotlib.legend.Legend at 0x7fd4447a45e0>
```

The learning curves should look something like this. Your learning curves may look a little different, but the tendencies should be the same:

 Learning curves for training the ANN

The loss and accuracy values get better and better for the training data. However, for the validation data, the loss seems to get worse again, while the accuracy remains within a narrow range. The values for training and validation data diverge, so we probably overfitted the model a little. We'll look at two ways to prevent this next.

Congratulations: You have created and trained your first deep learning model and slightly improved the accuracy of your model.

Limiting overfitting

As you saw, we slightly overfitted our model. Since we can't know the number of epochs we need in advance, this can happen quickly. It's especially annoying when our ANN takes a very long time to train and we have to fit it to the data again afterwards, just do we can specify fewer epochs for the training. There are two things we can do to avoid this: Regularization and **early stopping**.

You learned about regularization in the context of **Lasso** and **Ridge** in *Module 1, Chapter 1, Regularization*. Both add a term to the loss function internally, which should limit the number of features used. **Lasso** applies the L1 norm to the weights of the features and **Ridge** applies the L2 norm. In principle, we can do the same for the ANN by assigning one of the two norms to the **kernel_regularizer** parameter of **Dense**, as specified [here](#).

However, neural networks offer another kind of regularization, which you see very often in practice. It's called dropout. In this process, some randomly selected artificial neurons are deactivated at each learning step of the ANN. So they only return the value 0 and their weights

are not adjusted in this step. Overfitting is when the model learns the data by heart and some weights are very strongly defined. But this doesn't happen as much if some neurons are left out. As a result, the architecture of the neural network looks a little different at each learning step. This reduces the dependencies between the neurons. The entire network becomes more robust and can be better generalized to deal with data it hasn't yet seen. You should use the dropout method particularly when the training set is relatively small. Let's try out whether our ANN also gets better results when we use dropout.

In `tensorflow.keras` the dropout functionality is implemented as a separate layer. Imagine that the **dropout layer** has as many neurons as the layer before it. Each of its neurons gets the result of exactly one neuron from the normal layer. With a certain probability, the dropout neuron will either pass this value to the next layer or the value 0. So if we add a dropout layer after a `dense` layer, this means that some neurons of the `dense` layer will not output any values. The most important parameter for `Dropout` is `rate`. We pass a floating point number between 0 and 1.0 to this. It indicates the probability of a neuron being deactivated. Typically we use values between `0.2` and `0.5` here.

Now import `Dropout` from `tensorflow.keras.layers`. Then instantiate a new `Sequential` model under the name `model_ann_drop`. It should have the same architecture as `model_ann`, i.e. 5 hidden layers with 50 artificial neurons each and an output layer with one artificial neuron. But now add a dropout layer after each hidden layer. All the dropout layers should be given the parameter `rate=0.3`.

Tip: Instantiate `Dropout` in the variables `dropout_layer_first` to `dropout_layer_fifth`, with the appropriate `rate` value. Use the same hidden layers again - `hidden_first` to `hidden_fifth`, and `output_layer`.

```
In [15]: from tensorflow.keras.layers import Dropout
model_ann_drop = Sequential()
dropout_layer_first = Dropout(rate=0.3)
dropout_layer_second = Dropout(rate=0.3)
dropout_layer_third = Dropout(rate=0.3)
dropout_layer_fourth = Dropout(rate=0.3)
dropout_layer_fifth = Dropout(rate=0.3)

model_ann_drop.add(hidden_first)
model_ann_drop.add(dropout_layer_first)

model_ann_drop.add(hidden_second)
model_ann_drop.add(dropout_layer_second)

model_ann_drop.add(hidden_third)
model_ann_drop.add(dropout_layer_third)

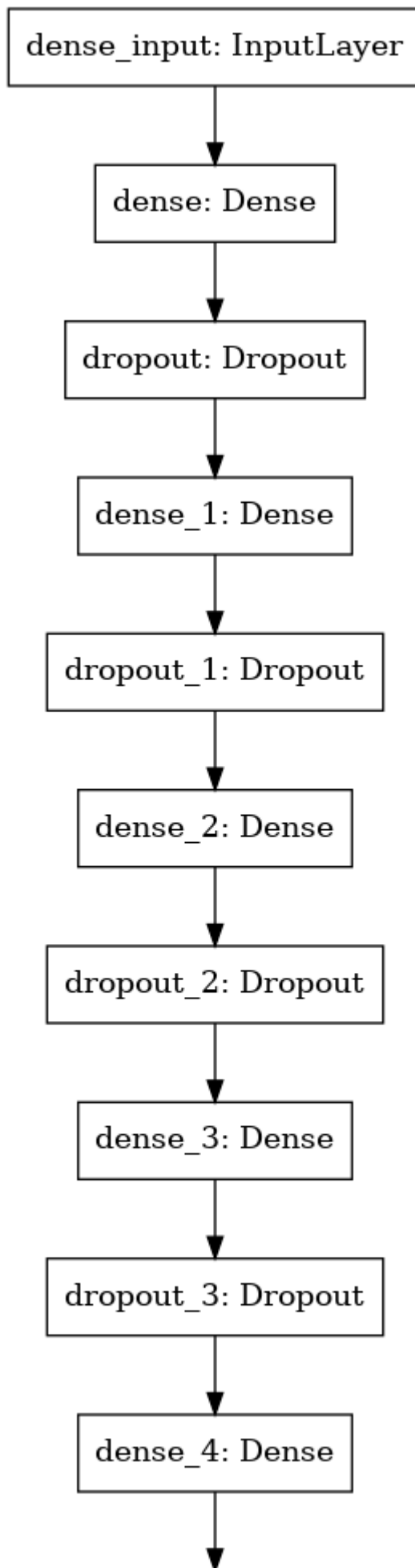
model_ann_drop.add(hidden_fourth)
model_ann_drop.add(dropout_layer_fourth)

model_ann_drop.add(hidden_fifth)
model_ann_drop.add(dropout_layer_fifth)
```

Now you have to compile the model again. Run the following code cell, which will generate a visualization directly.

```
In [16]: model_ann_drop.compile(optimizer = "adam", loss = 'binary_crossentropy', metrics = ['a  
plot_model(model_ann_drop)
```

Out[16]:



dropout_4: Dropout

If you use the Dropout method, it usually means that you need more epochs for the training. Since we have already used too many epochs anyway, we can reuse the same number. Use the following cell to train `model_ann_drop`. This may take a few minutes.

```
In [17]: hist_ann_drop = model_ann_drop.fit(features_train_scaled, target_train, epochs=20, bat
```

Epoch 1/20
4327/4327 [=====] - 8s 2ms/step - loss: 4.0236 - accuracy: 0.3178 - val_loss: 2.6104 - val_accuracy: 0.0618
Epoch 2/20
4327/4327 [=====] - 7s 2ms/step - loss: 3.0194 - accuracy: 0.2375 - val_loss: 2.6146 - val_accuracy: 0.0930
Epoch 3/20
4327/4327 [=====] - 7s 2ms/step - loss: 2.9287 - accuracy: 0.2823 - val_loss: 2.5977 - val_accuracy: 0.0000e+00
Epoch 4/20
4327/4327 [=====] - 8s 2ms/step - loss: 2.9336 - accuracy: 0.1399 - val_loss: 2.6110 - val_accuracy: 0.0000e+00
Epoch 5/20
4327/4327 [=====] - 8s 2ms/step - loss: 2.9129 - accuracy: 0.1641 - val_loss: 2.5953 - val_accuracy: 0.0097
Epoch 6/20
4327/4327 [=====] - 8s 2ms/step - loss: 2.9131 - accuracy: 0.1371 - val_loss: 2.5980 - val_accuracy: 4.5546e-04
Epoch 7/20
4327/4327 [=====] - 7s 2ms/step - loss: 2.9244 - accuracy: 0.1556 - val_loss: 2.5934 - val_accuracy: 0.0951
Epoch 8/20
4327/4327 [=====] - 8s 2ms/step - loss: 2.9247 - accuracy: 0.1577 - val_loss: 2.5957 - val_accuracy: 0.0843
Epoch 9/20
4327/4327 [=====] - 7s 2ms/step - loss: 2.9251 - accuracy: 0.1424 - val_loss: 2.5965 - val_accuracy: 0.0000e+00
Epoch 10/20
4327/4327 [=====] - 7s 2ms/step - loss: 2.9262 - accuracy: 0.2445 - val_loss: 2.5932 - val_accuracy: 0.4917
Epoch 11/20
4327/4327 [=====] - 7s 2ms/step - loss: 2.9271 - accuracy: 0.2291 - val_loss: 2.5954 - val_accuracy: 0.3976
Epoch 12/20
4327/4327 [=====] - 8s 2ms/step - loss: 2.9170 - accuracy: 0.1934 - val_loss: 2.5962 - val_accuracy: 0.4068
Epoch 13/20
4327/4327 [=====] - 7s 2ms/step - loss: 2.9274 - accuracy: 0.2247 - val_loss: 2.5981 - val_accuracy: 0.8586
Epoch 14/20
4327/4327 [=====] - 8s 2ms/step - loss: 2.9297 - accuracy: 0.3631 - val_loss: 2.6005 - val_accuracy: 0.8576
Epoch 15/20
4327/4327 [=====] - 7s 2ms/step - loss: 2.9258 - accuracy: 0.3927 - val_loss: 2.5991 - val_accuracy: 0.7735
Epoch 16/20
4327/4327 [=====] - 7s 2ms/step - loss: 2.9348 - accuracy: 0.3731 - val_loss: 2.6052 - val_accuracy: 0.3979
Epoch 17/20
4327/4327 [=====] - 7s 2ms/step - loss: 2.9168 - accuracy: 0.3405 - val_loss: 2.6071 - val_accuracy: 0.7735
Epoch 18/20
4327/4327 [=====] - 8s 2ms/step - loss: 2.9329 - accuracy: 0.3354 - val_loss: 2.5989 - val_accuracy: 0.8611
Epoch 19/20
4327/4327 [=====] - 8s 2ms/step - loss: 2.9206 - accuracy: 0.3379 - val_loss: 2.6010 - val_accuracy: 0.7735
Epoch 20/20
4327/4327 [=====] - 8s 2ms/step - loss: 2.9217 - accuracy: 0.3734 - val_loss: 2.6000 - val_accuracy: 0.7737

Now visualize the learning curves again. You can find the corresponding values in the dictionary `hist_ann.history` under the keys `'loss'`, `'val_loss'`, `'accuracy'` and `'val_accuracy'`.

```
In [18]: fig, axs = plt.subplots(ncols=2, figsize=(12,6))

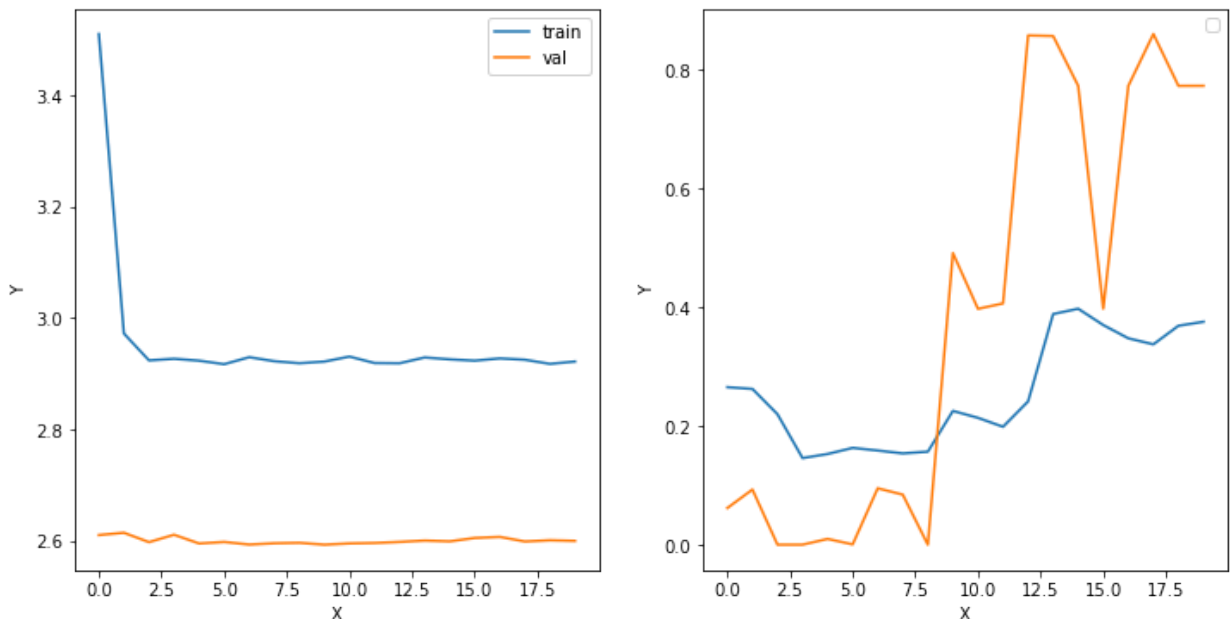
axs[0].plot(range(0,20),hist_ann_drop.history['loss'], label='train')
axs[0].plot(range( 0,20),hist_ann_drop.history['val_loss'], label='val')

axs[1].plot(range( 0,20),hist_ann_drop.history['accuracy'] )
axs[1].plot( range( 0,20), hist_ann_drop.history['val_accuracy'])


axs[0].set(xlabel='X', ylabel='Y')
axs[0].legend()
axs[1].set(xlabel='X', ylabel='Y')
axs[1].legend()
```

No handles with labels found to put in legend.

```
Out[18]: <matplotlib.legend.Legend at 0x7fd4086fe160>
```



You should get something like this:

 learning curves when training the ANN

The values of training and validation data no longer diverge. It's even possible that our model will improve a little if we continue training it. We can do this with the `my_model.fit()` method again. In contrast to `sklearn`, `keras` doesn't fit the model to the data starting from scratch, but it continues where you left off. We'll carry out another 10 epochs in the following cell:

```
In [19]: hist_ann_drop = model_ann_drop.fit(features_train_scaled, target_train, epochs=10, bat
```

```

Epoch 1/10
4327/4327 [=====] - 8s 2ms/step - loss: 2.9255 - accuracy:
0.4005 - val_loss: 2.5964 - val_accuracy: 0.7742
Epoch 2/10
4327/4327 [=====] - 7s 2ms/step - loss: 2.9261 - accuracy:
0.2544 - val_loss: 2.5992 - val_accuracy: 0.4915
Epoch 3/10
4327/4327 [=====] - 8s 2ms/step - loss: 2.9197 - accuracy:
0.3206 - val_loss: 2.5967 - val_accuracy: 0.3985
Epoch 4/10
4327/4327 [=====] - 7s 2ms/step - loss: 2.9285 - accuracy:
0.2969 - val_loss: 2.5948 - val_accuracy: 0.7913
Epoch 5/10
4327/4327 [=====] - 8s 2ms/step - loss: 2.9246 - accuracy:
0.2505 - val_loss: 2.5991 - val_accuracy: 0.7736
Epoch 6/10
4327/4327 [=====] - 7s 2ms/step - loss: 2.9221 - accuracy:
0.2426 - val_loss: 2.5990 - val_accuracy: 0.4777
Epoch 7/10
4327/4327 [=====] - 8s 2ms/step - loss: 2.9121 - accuracy:
0.2231 - val_loss: 2.5974 - val_accuracy: 0.3983
Epoch 8/10
4327/4327 [=====] - 7s 2ms/step - loss: 2.9191 - accuracy:
0.1956 - val_loss: 2.6034 - val_accuracy: 0.0822
Epoch 9/10
4327/4327 [=====] - 8s 2ms/step - loss: 2.9383 - accuracy:
0.2253 - val_loss: 2.6270 - val_accuracy: 0.4007
Epoch 10/10
4327/4327 [=====] - 7s 2ms/step - loss: 2.9308 - accuracy:
0.2394 - val_loss: 2.6004 - val_accuracy: 0.4569

```

We can see from the values that the accuracy continues to fluctuate around a high value close to 0.93. Further training does not seem to improve the model.

Another way to avoid overfitting the model too much is to stop training early. This procedure is called **early stopping**.

Run the following cell to define and build the model without dropout.

```

In [21]: # define model
model_ann_early = Sequential()

# define hidden layers
hidden_first = Dense(units=50, activation='relu', input_dim=features_train_scaled.shape[1])
hidden_second = Dense(units=50, activation='relu')
hidden_third = Dense(units=50, activation='relu')
hidden_fourth = Dense(units=50, activation='relu')
hidden_fifth = Dense(units=50, activation='relu')

# define output layer
output_layer = Dense(units=1, activation='sigmoid')

# add 5 hidden layers with 50 units
model_ann_early.add(hidden_first)
model_ann_early.add(hidden_second)
model_ann_early.add(hidden_third)
model_ann_early.add(hidden_fourth)
model_ann_early.add(hidden_fifth)

```

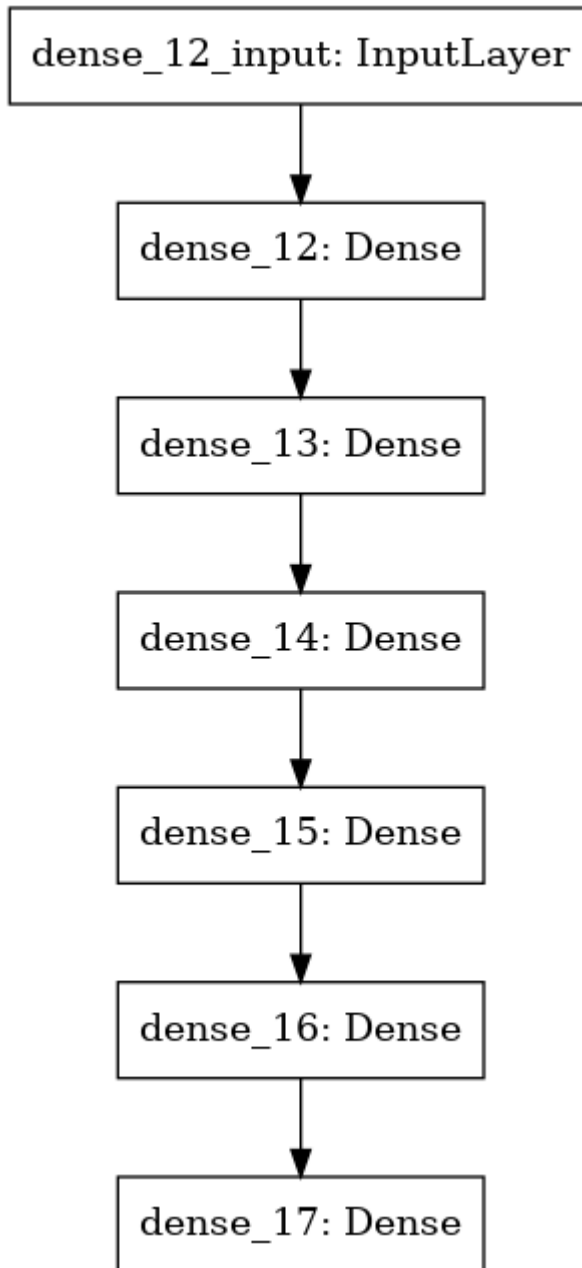


```
# add output layer
model_ann_early.add(output_layer)

# compile model
model_ann_early.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# plot the model
plot_model(model_ann_early)
```

Out[21]:



`tensorflow.keras` offers what are called [callback objects](#). We can pass these to the `my_model.fit()` method so that they are implemented during training. One of these is `EarlyStopping`. It stops training if a specified metric doesn't continue to improve. `EarlyStopping` has to be instantiated. You pass the name of the value to be monitored to the `monitor` parameter. Here we can use for example the accuracy of the validation data `'val_accuracy'`. The parameter `min_delta` indicates the minimum size of an improvement that has to be achieved in order to be considered an improvement. During training we could

see that 'val_accuracy' was fluctuating in the third decimal place. So we recommend a value of 0.001 to only react to real improvements. patience indicates how many epochs without improvement can happen before the training is stopped. Here, for example, we select the value 2.

Define early_stop as an instance of EarlyStopping from tensorflow.keras.callbacks with the parameters specified in the text.

```
In [22]: from tensorflow.keras.callbacks import EarlyStopping
early_stop = EarlyStopping(monitor='val_accuracy', min_delta=0.001, patience=2)
```

Now Let's train the model. Let's specify 200 epochs for the training duration. But at the same time, we'll give pass only_stop to the parameter callbacks in a list. How many epochs are actually used for training?

```
In [23]: # train model using early stopping
model_ann_early.fit(features_train_scaled, target_train,
                    epochs=200, batch_size=64,
                    callbacks=[early_stop],
                    validation_data=(features_val_scaled, target_val))
```

```
Epoch 1/200
4327/4327 [=====] - 11s 3ms/step - loss: 0.1641 - accuracy:
0.9177 - val_loss: 0.1334 - val_accuracy: 0.9288
Epoch 2/200
4327/4327 [=====] - 10s 2ms/step - loss: 0.1343 - accuracy:
0.9286 - val_loss: 0.1327 - val_accuracy: 0.9289
Epoch 3/200
4327/4327 [=====] - 11s 2ms/step - loss: 0.1317 - accuracy:
0.9296 - val_loss: 0.1319 - val_accuracy: 0.9292
<tensorflow.python.keras.callbacks.History at 0x7fd3e4416040>
```

Out[23]:

In our case, 5 epochs were carried out and the accuracy of the model is about 93.0% for the validation data. The exact values may differ slightly for you, but it should be significantly less than 200 epochs. We can't get much more accuracy out of this model easily. But this is a very good value for a first attempt! It's a good idea to save the model. This means in future we won't have to define and train this model again. tensorflow.keras makes it very easy for us to save a model by providing us with the my_model.save() method. We simply specify the save path and tensorflow.keras then generates an HDF5 file with the model's architecture and weights.

Run the cell to save the model:

```
In [24]: model_ann_early.save('model_ann.h5')
```

To load a model, tensorflow.keras.models offers the load_model function, which only requires the model's file path.

You might have noticed that in this case the ANN is not much better than logistic regression. ANNs don't always produce impressive results. They usually need a lot of data to be well

trained. Training takes a long time in comparison and they have a lot of hyperparameters due to all the individual layers. Fine tuning the hyperparameters is an art in itself. The topic of ANNs and their architectures is so extensive that you could devote an entire training course to it. For this reason, it's often advisable to use one of the other models you learned about in this module.

But everyone's talking about ANNs for a reason, so even management at *Lemming Loans Limited* wanted to use this technology. Their great strength is that they can independently recognize very complicated structures in the data. This strength is clearly when it comes to image classification, for example. Sometimes artificial neural networks surpass other models very clearly. Once you've found a good architecture and trained the ANN has been trained for a long time with a lot of good data, you can start to see results. Fortunately there are already many saved models that are freely available. `keras` offers some of them [here](#).

Congratulations: You now know how to limit or even prevent overfitting with a neural network. Your superiors at *Lemming Loans Limited* are very pleased with you and say thank you for your work!

Remember:

- Creating and training deep neural networks can be very time consuming.
- You can use the training history to evaluate if your model is overfitted.
- `EarlyStopping` stops the training and can save a lot of computing time.
- `Dropout` regularizes the ANN and makes it more robust.

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

Found a mistake? Contact Support at support@stackfuel.com.