IntroToNeuralNetworks - IntroToTensorflow - 2

One should look for what is and not what he thinks should be. (Albert Einstein)

Module completion checklist

Objective	Complete
Prepare data for implementing neural network	
Implement and evaluate model on test data	

Goal

- In this section we will build a **neural network with 2 hidden layers** in TensorFlow using the `tf.keras` module
- We will create this network using a **sequential** model
- To do this, we will be using the `Credit Card` dataset

Coding overview

Specifically, we will:

1. Clean and wrangle the dataset so it is suitable for the neural network model
2. Split the dataset into train, test, and validation data
3. Define and compile the sequential model
4. Fit the model on training data
5. Compare the training/validation accuracy for each epoch
6. Evaluate and predict on test data

Before we start, let's go through a few quick coding notes

Coding in TensorFlow and Keras

- Whenever we intend to use TensorFlow's native module, we will follow this syntax:
  - `tf.[module_name]`
- When we intend to use Keras, we will will follow this syntax:
  - `tf.keras.[module_name]`
- Or, we can also import specific keras modules separately and use them directly:
  - `from tensorflow.keras.layers import Dense`
  - then just use `Dense` in our code

Loading packages

- Let's load the packages we will be using:

```
import os
import pickle
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import math
import seaborn as sns
# Scikit-learn packages.
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn import metrics
# TensorFlow and supporting packages.
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier
from tensorflow.keras.models import load_model
from tensorflow.keras.optimizers import Adam
```

Directory settings

- In order to maximize the efficiency of your workflow, you should encode your directory structure into `variables`
- We will use the `pathlib` library
- Let the `main_dir` be the variable corresponding to your course materials folder
- `data_dir` be the variable corresponding to your `data` folder

```
# Set 'main_dir' to location of the project folder
from pathlib import Path
home_dir = Path(".").resolve()
main_dir = home_dir.parent.parent
print(main_dir)
```

```
data_dir = str(main_dir) + "/data"
print(data_dir)
```

Load the data

- The `credit_card_data` dataset contains information about credit card **defaulters**
- Our goal is to predict if the customer will **default on a credit card payment or not**

```
credit_card = pd.read_csv(str(data_dir) + '/credit_card_data.csv')
print(credit_card.head())
```

```
   ID  LIMIT_BAL  SEX  ...  PAY_AMT5  PAY_AMT6  default_payment_next_month
0   1      20000    2  ...         0         0                          1
1   2     120000    2  ...         0      2000                          1
2   3      90000    2  ...      1000      5000                          0
3   4      50000    2  ...      1069      1000                          0
4   5      50000    1  ...       689       679                          0

[5 rows x 25 columns]
```

Data preparation

- Before starting to implement our neural network, we need to make sure our data is clean and is in the right form, for that we need to:
  - check the data for **NAs**
  - **transform the data to numeric values** - if it's categorical, make sure the data is encoded
  - split data into train, test, and validation
  - normalize data
  - examine the target variable imbalance

Note: the order of operations matters! Ideally, all data transformations should happen after the data has been split. In this instance we will check for NAs and encode categorical variables before the split, since this will not greatly affect the results and keep our code more concise.

Data prep: convenience function

- Lucky for you, Data Society wrote a time-saving function to perform all of the cleaning and split steps on the credit card dataset at once!

```
def data_prep(df):
    df = df.fillna(df.mean()['BILL_AMT1'])
    df = df.drop('ID',axis = 1)
    # Convert 'sex' into dummy variables.
    sex = pd.get_dummies(df['SEX'], prefix = 'sex', drop_first = True)
    # Convert 'education' into dummy variables.
    education = pd.get_dummies(df['EDUCATION'], prefix = 'education', drop_first = True)
    # Convert 'marriage' into dummy variables.
    marriage = pd.get_dummies(df['MARRIAGE'], prefix = 'marriage', drop_first = True)
    # Drop `sex`, `education`, `marriage` from the data.
    df.drop(['SEX', 'EDUCATION', 'MARRIAGE'], axis = 1, inplace = True)
    # Concatenate `sex`, `education`, `marriage` dummies to our dataset.
    df = pd.concat([df, sex, education, marriage], axis=1)
    # Separate predictors from data.
    X = df.drop(['default_payment_next_month'], axis=1)
    y = df['default_payment_next_month']
```

Data prep: convenience function (cont'd)

```
    # Set the seed to 1.
    np.random.seed(1)
    # Split data into train, test, and validation set, use a 70 - 15 - 15 split.
    # First split data into train-test with 70% for train and 30% for test.
    X_train, X_test, y_train, y_test = train_test_split(X.values,
                                                        y,
                                                        test_size = .3,
                                                        random_state = 1)
    # Then split the test data into two halves: test and validation.
    X_test, X_val, y_test, y_val = train_test_split(X_test,
                                                    y_test,
                                                    test_size = .5,
                                                    random_state = 1)
```

Data prep: convenience function (cont'd)

```
    print("Train shape:", X_train.shape, "Test shape:", X_test.shape, "Val shape:", X_val.shape)

    # Transforms features by scaling each feature to a given range.
    # The default is the range between 0 and 1.
    min_max_scaler = preprocessing.MinMaxScaler()
    X_train_scaled = min_max_scaler.fit_transform(X_train)
    X_test_scaled = min_max_scaler.transform(X_test)
    X_val_scaled = min_max_scaler.transform(X_val)
    return X_train_scaled, X_test_scaled, X_val_scaled, y_train, y_test, y_val
```

Data prep

```
X_train_scaled, X_test_scaled, X_val_scaled, y_train, y_test, y_val = data_prep(credit_card)
```

```
Train shape: (21000, 30) Test shape: (4500, 30) Val shape: (4500, 30)
```

Module completion checklist

Objective	Complete
Prepare data for implementing neural network	✔
Implement and evaluate model on test data	

Steps to define your model

Define the input layer and ensure that either:

1. `input_dim` from `Dense` is specified to match the number of inputs to the model *or*
2. `input_shape` is a tuple of integers, or a shape tuple, where the batch dimension is not included *or*
3. Pass a list of TF-compatible feature columns, which have the shape, dimensions and data type specified

Determine the number of layers for your model:

1. This is usually done through trial and error based on the heuristics of your model
2. You can start with the idea that you need a network large enough to capture the structure of the problem

Steps to define your model (cont'd)

Use `Dense` to define each layer, specifying:

1. Number of neurons in the given layer as the first argument
2. Activation function using the activation argument

Define the output layer:

1. Number of units to predict (`1` if binary)
2. Choose the activation function for the output, sigmoid if binary

Implement Sequential model with Dense layers

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

# Let's set the seed so that we can reproduce the results.
tf.random.set_seed(1)

## Set up model.
model = Sequential([
  Dense(32,                                          #<- number of neurons for 1st hidden layer
        input_shape = (X_train_scaled.shape[1], )), #<- input layer shape: `(num_features, )`
  Dense(20,                                          #<- add 2nd hidden layer with 20 neurons
        activation = 'relu'), #<- set activation function for hidden layer
  Dense(1,                    #<- add output layer with single neuron
       activation = 'sigmoid')#<- activation function for output layer
])
```

Compile the model

- Once the model is initialized, we need to **compile and fit** it
- There are some parameters that we have to choose at this stage:
  - **loss function**: function we use to evaluate the set of weights, in our case it will be the already familiar to us `"binary_crossentropy"`
  - **gradient descent algorithm**: optimization algorithm, in our case `"adam"`, it is popular and efficient
  - **metrics**: the metrics we want as output, let's stick to a simple one `"accuracy"`

```
# Compile the model.
model.compile(optimizer = "adam",
              loss = "binary_crossentropy",
              metrics = ["accuracy"])
```

- You can review ***loss function***, ***optimization algorithm***, and ***metrics*** documentation using the links provided

Fit the model

- Finally, we have what we need to **fit the model** to our data
- The training process will run for a fixed number of iterations ("epochs") through the dataset
- We specify this using the `epochs` argument
- We can also add validation data to compare train and validation metrics and loss

```
model_res = model.fit(X_train_scaled, y_train,                  #<- train data and labels
                      validation_data = (X_val_scaled, y_val), #<- pass val data
                      epochs = 200)                            #<- number of epochs
```

```
Epoch 1/200
657/657 [==============================] - 1s 1ms/step - loss: 0.5129 - accuracy: 0.7841 -
val_loss: 0.4606 - val_accuracy: 0.8049
Epoch 2/200
657/657 [==============================] - 1s 884us/step - loss: 0.4581 - accuracy: 0.8099 -
val_loss: 0.4555 - val_accuracy: 0.8091
...
Epoch 199/200
657/657 [==============================] - 1s 841us/step - loss: 0.4161 - accuracy: 0.8230 -
val_loss: 0.4345 - val_accuracy: 0.8224
Epoch 200/200
657/657 [==============================] - 1s 838us/step - loss: 0.4152 - accuracy: 0.8273 -
val_loss: 0.4328 - val_accuracy: 0.8211
```

Model evaluation metrics

- We will evaluate our model based on the following metrics:
  - Training/validation **accuracy** for each epoch
  - Training/validation **loss** for each epoch

- At the end of each epoch, the **loss** is calculated by comparing the model's predictions with the original labels
- This is used to update the loss function each time and the weights are adjusted accordingly for every epoch
- **Accuracy**: The ratio of total records being classified correctly to the total number of records in the dataset

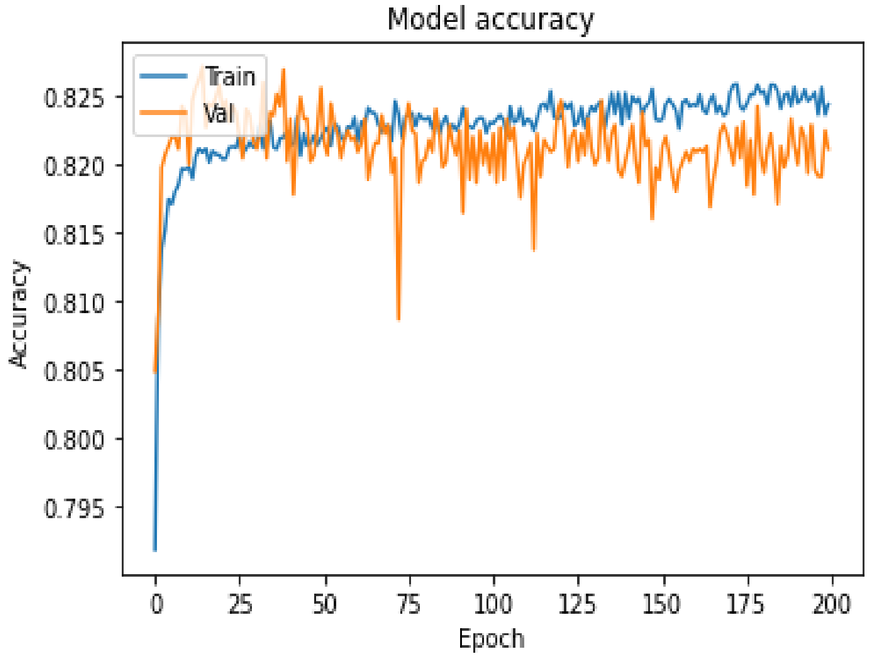
Model evaluation metrics (cont'd)

- In general, the model is said to perform well if it gives good accuracy with less amount of loss
- To assess the neural network model, we can quickly visualize and observe the pattern of accuracy and loss for each epoch

Visualize training/validation accuracy for each epoch

- The `model_res` object contain model results in a `history` dictionary
- You can access accuracy for train data by calling `model_res.history['accuracy']`
- You can access accuracy for validation data by calling `model_res.history['val_accuracy']`

```
# Plot training & validation accuracy values
plt.plot(model_res.history['accuracy'])    #<- accuracy
scores
plt.plot(model_res.history['val_accuracy']))#<- get val
accuracy scores from dictionary
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Val'], loc='upper left')
plt.show()
```

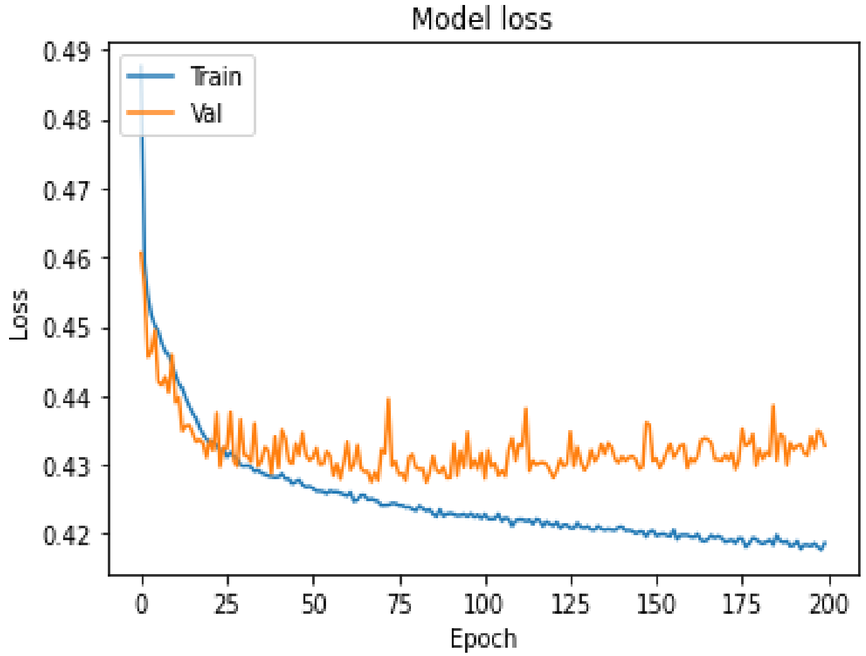


Overall, the training and the validation accuracy seem to be increasing as the number of epochs increase!

Visualize training/validation loss for each epoch

- You can access loss values for train data by calling `model_res.history['loss']`
- You can access loss values for validation data by calling `model_res.history['val_loss']`

```
# Plot training & validation loss values
plt.plot(model_res.history['loss'])
plt.plot(model_res.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Val'], loc='upper left')
plt.show()
```



Evaluate loss, accuracy on test data and predict

```
loss, accuracy = model.evaluate(x = X_test_scaled, y = y_test)
```

```
141/141 [==============================] - 0s 611us/step - loss: 0.4448 - accuracy: 0.8127
```

```
print("Loss: {0:6.3f}, Accuracy: {1:6.3f}".format(loss, accuracy))
```

```
Loss:  0.445, Accuracy:  0.813
```

```
y_pred_prob = model.predict(X_test_scaled)
y_pred = (model.predict(X_test_scaled) > 0.5).astype("int32")
print(y_pred)
```

```
[[1]
 [0]
 [0]
 ...
 [1]
 [0]
 [0]]
```

Knowledge check

Link: ***https://forms.gle/nDCVm59yVcDixfNo6***

Exercise

You are now ready to try Tasks 1-7 in the Exercise for this topic

Module completion checklist

Objective	Complete
Prepare data for implementing neural network	✔
Implement and evaluate model on test data	✔

Intro To TensorFlow: Topic summary

In this part of the course, we have covered:

- Overview of TensorFlow / Keras building blocks
- Implement and fit a neural network model using Tensorflow on train data
- Evaluating neural network model on test data

Congratulations on completing this module!