# IntroToNeuralNetworks - BuildingNeuralNetworks - 1

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

# Building Neural Networks: Topic introduction

In this part of the course, we will cover the following concepts:

- Create a basic neural network model
- Evaluating models using various performance metrics
- Visualize accuracy and loss

# Module completion checklist

| Objective | Complete |
|---|---|
| Identify data processing steps and prepare data for analysis | |
| Introduce MLPClassifier for building a simple neural network | |

# Warm up: What is a neural network?

- Watch this video about **Neural Networks**, found ***here***
- This 5 minute video should recap **Neural Networks** and their applications
- After watching the video, reflect on three applications of **Neural Networks** that are encountered in day to day life

# Datasets for this module

- The first step to building a simple NN is to prepare the data
- We will be using two datasets in this module:
    - One to learn the concepts in class: **Credit Card data**
    - One for our in-class exercises: **Bank Marketing data**

# 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)
```

# Loading packages

- These are the packages we will use for data wrangling and creating a simple neural network

```
# Helper packages.
import os
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pickle
from textwrap import wrap
```

```
# Scikit-learn package for building a perceptron.
from sklearn.neural_network import MLPClassifier

# Scikit-learn package for data preprocessing.
from sklearn.preprocessing import MinMaxScaler
```

```
# Model set up, tuning and model metrics packages.
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn import preprocessing
from sklearn import metrics
from sklearn.model_selection import GridSearchCV
```

# Load the data

- We are going to load the `credit_card_data` dataset
- This dataset contains information about credit card **defaulters**
- The 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 cleaning

- We need to make sure our data is in the right form to run through a neural network, which is why we must:
    - check the data for NAs
    - encode categorical data into numeric
    - split data into train and test sets
    - scale data
    - target balance the train set (if target variable is not balanced)

**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.

# The data at first glance

- Look at the data types of each variable

```
# The data types.
print(credit_card.dtypes)
```

```
ID                                     int64
LIMIT_BAL                              int64
SEX                                    int64
EDUCATION                              int64
MARRIAGE                               int64
AGE                                    int64
PAY_0                                  int64
PAY_2                                  int64
PAY_3                                  int64
PAY_4                                  int64
PAY_5                                  int64
PAY_6                                  int64
BILL_AMT1                            float64
BILL_AMT2                              int64
BILL_AMT3                              int64
BILL_AMT4                              int64
BILL_AMT5                              int64
BILL_AMT6                              int64
```

# Check for NAs in the dataset

- Check for NAs

```
# Check for NAs.
print(credit_card.isnull().sum())
```

```
ID              0
LIMIT_BAL       0
SEX             0
EDUCATION       0
MARRIAGE        0
AGE             0
PAY_0           0
PAY_2           0
PAY_3           0
PAY_4           0
PAY_5           0
PAY_6           0
BILL_AMT1       1
BILL_AMT2       0
BILL_AMT3       0
BILL_AMT4       0
BILL_AMT5       0
BILL_AMT6       0
```

- We have `1` missing value in the variable column `BILL_AMT1`
- We can either **impute** missing values or **drop** them to prepare the dataset for a neural network model
  - We will **impute** the missing value to demonstrate how it works
  - You can also simply **drop** missing values;
- **Note:** If the dataset contains a lot of missing values you will lose a lot of observations due to that!

# Using fillna() to handle missing values

- We will fill the missing value in `BILL_AMT1` with the mean of the column

```
 # Fill missing values with mean
credit_card = credit_card.fillna(credit_card.mean()['BILL_AMT1'])
```

```
 # Check for NAs in 'BILL_AMT1'.
print(credit_card.isnull().sum()['BILL_AMT1'])
```

```
0
```

- We see that there aren't any NAs in the dataset anymore
- Let's drop the unnecessary identifiers from the dataset

```
# Drop an unnecessary identifier column.
credit_card = credit_card.drop('ID',axis = 1)
```

# Dummy variables: one hot encoding

- A dummy variable is an **artificial variable** used to represent a variable with **two or more distinct levels or categories**
- It represents categorical predictors as binary values, **0 or 1**

| ID | Pet |
|---|---|
| 1 | Dog |
| 2 | Cat |
| 3 | Cat |
| 4 | Dog |
| 5 | Fish |

| ID | Dog | Cat | Fish |
|---|---|---|---|
| 1 | 1 | 0 | 0 |
| 2 | 0 | 1 | 0 |
| 3 | 0 | 1 | 0 |
| 4 | 1 | 0 | 0 |
| 5 | 0 | 0 | 1 |

# Dummy variables: reference category

- The number of dummy variables necessary to represent a single attribute variable is equal to the **number of levels (categories) in that variable minus one**
- One of the categories is omitted and used as a **base or reference category**
- The reference category, which is not coded, is the category to which **all other categories will be compared**
- The biggest group / category will often be the reference category

# Dummy variables in Python

```
pd.get_dummies(dataframe['Column'],
               drop_first = False,
               ...)
```

- `data` is a pandas `Series` or `Dataframe`
- `drop_first` indicates whether to get `k-1` dummies out of `k` categorical levels

## pandas.get_dummies

`pandas.get_dummies`(*data*, *prefix=None*, *prefix_sep='_'*, *dummy_na=False*, *columns=None*, *sparse=False*, *drop_first=False*, *dtype=None*) [source]

Convert categorical variable into dummy/indicator variables

**Parameters:**

**data** : *array-like, Series, or DataFrame*

**prefix** : *string, list of strings, or dict of strings, default None*
String to append DataFrame column names. Pass a list with length equal to the number of columns when calling get_dummies on a DataFrame. Alternatively, *prefix* can be a dictionary mapping column names to prefixes.

**prefix_sep** : *string, default '_'*
If appending prefix, separator/delimiter to use. Or pass a list or dictionary as with *prefix*.

**dummy_na** : *bool, default False*
Add a column to indicate NaNs, if False NaNs are ignored.

**columns** : *list-like, default None*
Column names in the DataFrame to be encoded. If *columns* is None then all the columns with *object* or *category* dtype will be converted.

**sparse** : *bool, default False*
Whether the dummy-encoded columns should be be backed by a `SparseArray` (True) or a regular NumPy array (False).

**drop_first** : *bool, default False*
Whether to get k-1 dummies out of k categorical levels by removing the first level.
*New in version 0.18.0.*

**dtype** : *dtype, default np.uint8*
Data type for new columns. Only a single dtype is allowed.
*New in version 0.23.0.*

**Returns:**

**dummies** : *DataFrame*

# Transform and replace categorical variables

- Let's transform the categorical values into dummy variables and save it into a dataframe

```
# Convert 'sex' into dummy variables.
sex = pd.get_dummies(credit_card['SEX'], prefix = 'sex', drop_first = True)
# Convert 'education' into dummy variables.
education = pd.get_dummies(credit_card['EDUCATION'], prefix = 'education', drop_first = True)
# Convert 'marriage' into dummy variables.
marriage = pd.get_dummies(credit_card['MARRIAGE'], prefix = 'marriage', drop_first = True)
```

```
# Drop `sex`, `education`, `marriage` from the data.
credit_card.drop(['SEX', 'EDUCATION', 'MARRIAGE'], axis = 1, inplace = True)
```

```
# Concatenate `sex`, `education`, `marriage` dummies to our dataset.
credit_card = pd.concat([credit_card, sex, education, marriage], axis=1)
print(credit_card.head())
```

```
   LIMIT_BAL  AGE  PAY_0  ...  marriage_1  marriage_2  marriage_3
0      20000   24      2  ...           1           0           0
1     120000   26     -1  ...           0           1           0
2      90000   34      0  ...           0           1           0
3      50000   37      0  ...           1           0           0
4      50000   57     -1  ...           1           0           0

[5 rows x 31 columns]
```

# Data prep: split

```
# Separate predictors from data.
X = credit_card.drop(['default_payment_next_month'], axis=1)

# Separate target from data.
y = credit_card['default_payment_next_month']
```

- The common practice is to keep about 70-80% of your data for model training and split the remaining data in half
- We will stick to a simpler approach and split the data into train and test sets using a 70/30 rule

```
# Split data into train and test set, use a 70 - 30 split.
X_train, X_test, y_train, y_test = train_test_split(X.values,
                                                    y,
                                                    test_size = .3,
                                                    random_state = 1)

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

```
Train shape: (21000, 30) Test shape: (9000, 30)
```

- **Note:** remember to fix your random seed before you split the data, so that you can reproduce your results!

# Data prep: scale with MinMaxScaler

- NNs are sensitive to data scale
- There are a few methods to scale data
- We will use scikit-learn's `MinMaxScaler`

- **Note:** Scale values for train and test datasets separately to avoid data linkage that may introduce bias!

## sklearn.preprocessing.MinMaxScaler¶

*class* sklearn.preprocessing. **MinMaxScaler** (*feature_range=(0, 1)*, *copy=True*) [source]

Transforms features by scaling each feature to a given range.

This estimator scales and translates each feature individually such that it is in the given range on the training set, e.g. between zero and one.

The transformation is given by:

```
X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
X_scaled = X_std * (max - min) + min
```

where min, max = feature_range.

The transformation is calculated as:

```
X_scaled = scale * X + min - X.min(axis=0) * scale
where scale = (max - min) / (X.max(axis=0) - X.min(axis=0))
```

This transformation is often used as an alternative to zero mean, unit variance scaling.

```
# Transforms 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)
```

# Module completion checklist

| Objective | Complete |
|---|---|
| Identify data processing steps and prepare data for analysis | ✔ |
| Introduce MLPClassifier for building a simple neural network | |

# scikit-learn - MLPClassifier

- We will be using the `neural_network.MLPClassifier` module from `scikit-learn` package

**sklearn.neural_network.MLPClassifier**

*class* sklearn.neural_network.**MLPClassifier**(*hidden_layer_sizes=(100,), activation='relu', *, solver='adam', alpha=0.0001, batch_size='auto', learning_rate='constant', learning_rate_init=0.001, power_t=0.5, max_iter=200, shuffle=True, random_state=None, tol=0.0001, verbose=False, warm_start=False, momentum=0.9, nesterovs_momentum=True, early_stopping=False, validation_fraction=0.1, beta_1=0.9, beta_2=0.999, epsilon=1e-08, n_iter_no_change=10, max_fun=15000*) [source]

Multi-layer Perceptron classifier.

- The input is two arrays:
  - `X` which is a sparse or dense matrix that holds `[n_samples, n_features]`, and contains the training samples
  - `y` which is a vector of integers that holds `[n_samples]`, and contains the class labels for the training samples
- For all the parameters of the `MLPClassifier` package, visit ***scikit-learn's documentation***

# scikit-learn - MLPClassifier (cont'd)

- MLP stands for *multi-layer perceptron* model
- We will first use it as a single layer perceptron
- Two important factors in the performance of your model are:
    - **number of hidden layers**
    - **number of hidden neurons within each layer**
- As a rule of thumb, remember that too many or too few of either of these will affect your model's fit

# scikit-learn - MLPClassifier (cont'd)

- More hidden layers and hidden neurons will generally improve your model; however, this could lead to overfitting problems
- If you end up removing too many of either, this could underfit your model
- Neural networks are also used for regression type problems, where the target is continuous
  - In that case, you should use ***MLPRegressor*** that is also within scikit-learn's neural network library

# Implementing MLPClassifier in three steps

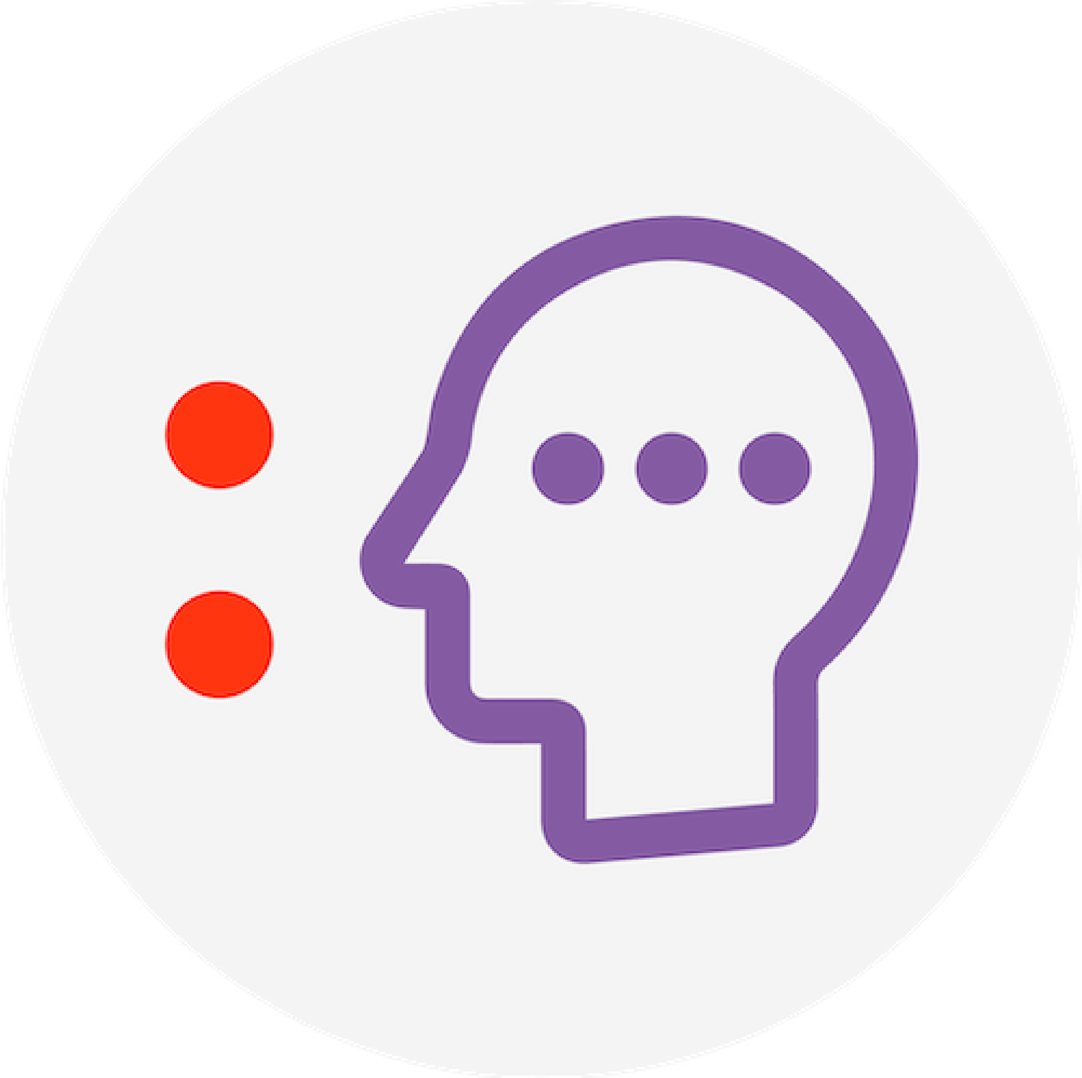
- To build a single layer perceptron classifier on our clean data we need to apply 3 methods

**Methods**

| | |
|---|---|
| `fit` (X, y) | Fit the model to data matrix X and target(s) y. |
| `get_params` ([deep]) | Get parameters for this estimator. |
| `predict` (X) | Predict using the multi-layer perceptron classifier |
| `predict_log_proba` (X) | Return the log of probability estimates. |
| `predict_proba` (X) | Probability estimates. |
| `score` (X, y[, sample_weig | Returns the mean accuracy on the given test data and labels. |
| `set_params` (**params) | Set the parameters of this estimator. |

- We need to:
  - **build** the model architecture
  - **fit** the model to the training data
  - **predict** on the test data using our trained model
- We'll dive deeper into each of these steps in the next module!

# Knowledge check



Link: **https://forms.gle/ZCjNVkZSW4gfEvL97**

# Module completion checklist

| Objective | Complete |
| --- | --- |
| Identify data processing steps and prepare data for analysis | ✔ |
| Introduce MLPClassifier for building a simple neural network | ✔ |

# Congratulations on completing this module!

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