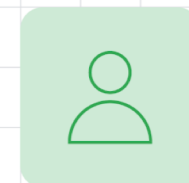


Google Developers



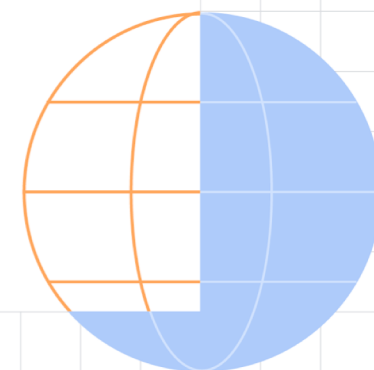
Deep Learning for Tabular Data

Integrating TensorFlow/Keras
and Scikit-learn worlds

GDE program



Luca Massaron
GDE in Machine Learning
@lmassaron



Time to start



Agenda

- Challenges for DNN for tabular data
- Exploring the data pipeline
- Cross validation as the right approach
- Wrapping up with generators and `tf.data`
- A working example
- Motivations for using DNN for tabular data

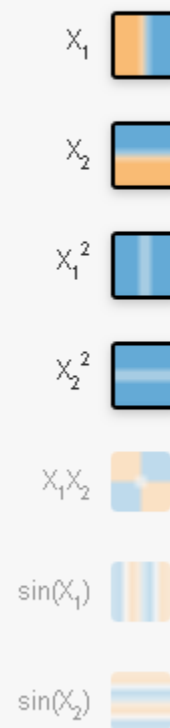
A short recap on the topic

- Deep learning is a **subset of machine learning**, which in turn is a **subset of artificial intelligence (AI)**.
- Deep learning is just a subset of AI, but it is an important subset. You see deep learning used for a wide number of tasks, but not every task.
- **Deep Neural Networks (DNN)** are the algorithms behind deep learning.

How a DNN works

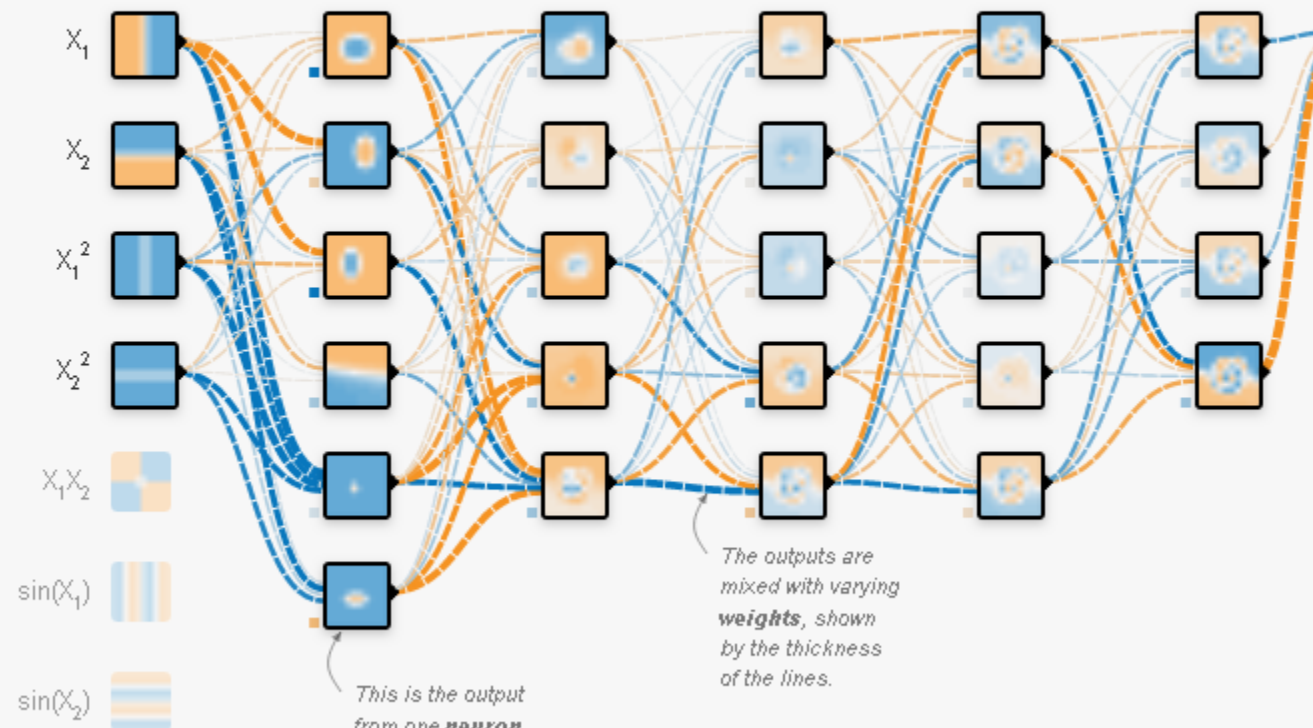
FEATURES

Which properties do you want to feed in?



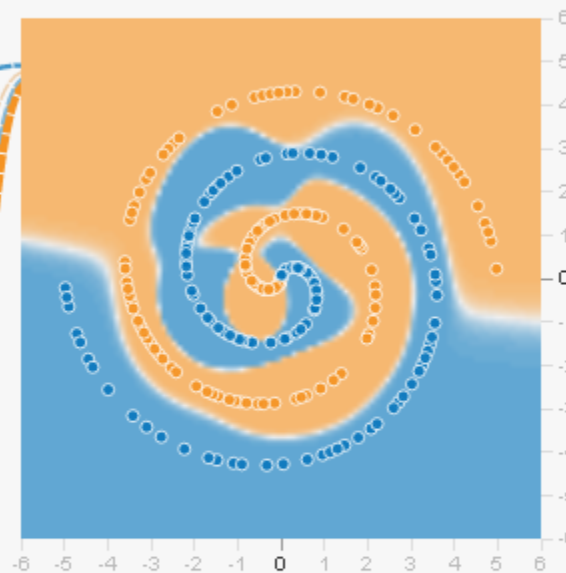
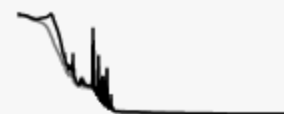
5 HIDDEN LAYERS

6 neurons 5 neurons 5 neurons 5 neurons 4 neurons

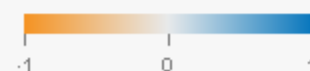


OUTPUT

Test loss 0.003
Training loss 0.005



Colors shows data, neuron and weight values.



☐ Show test data ☐ Discretize output

<https://playground.tensorflow.org/>

Why it is relevant?

- Machine learning, and especially deep learning, is associated with the idea of **software 2.0** (see *Andrej Karpathy*, senior director AI, Tesla): in such a paradigm you have **software not programmatically defined, but taught by examples.**

Great advances in DNNs

- Self-driving cars
- Deep learning in healthcare
- Deep learning in search engines
- Voice Search & Voice-Activated Assistants
- Automatic Machine Translation
- Sophisticated recommender systems
- Image generation and manipulation
- ...

But most data are not images or text

Relational Model

Activity Code	Activity Name
23	Patching
24	Overlay
25	Crack Sealing

Key = 24

Activity Code	Date	Route No.
24	01/12/01	I-95
24	02/08/01	I-66

Date	Activity Code	Route No.
01/12/01	24	I-95
01/15/01	23	I-495
02/08/01	24	I-66

Images or text are not the most frequent data you handle. Since long, **relational databases** have fostered the storage a mix of **numeric, symbolic and textual data**, scattered all together through tables.

Challenges of tabular data

- **Mixed features** data types
- **Sparse data** which is not the best for DNN
- **No SOTA/ best practice** architecture
- **Less data** than in image recognition problems
- There's suspect from non technical people because DNN are **less interpretable** than simpler ML algorithms
- **Often no best in class solution**, because GBM might perform better

But don't get discouraged

- **The challenges are serious, so are the opportunities**
- Andrew Ng: *“Deep learning has seen tremendous adoption in consumer Internet companies with a huge number of users and thus big data, but for it to break into other industries where datasets sizes are smaller, we now need better techniques for small data”*



To successfully use DNNs for tabular

- It takes some **effort**, don't expect an automated process or great results at once
- Don't re-invent the wheel: use **TensorFlow/Keras, Scikit-learn, Pandas** for your project
- Process and pipeline input accordingly to its **type**
- Create a **suitable neural architecture** keeping into account the number of available examples
- Encode prior knowledge (**feature engineering**)
- Test and tune your network using **cross validation**

Use the right tools



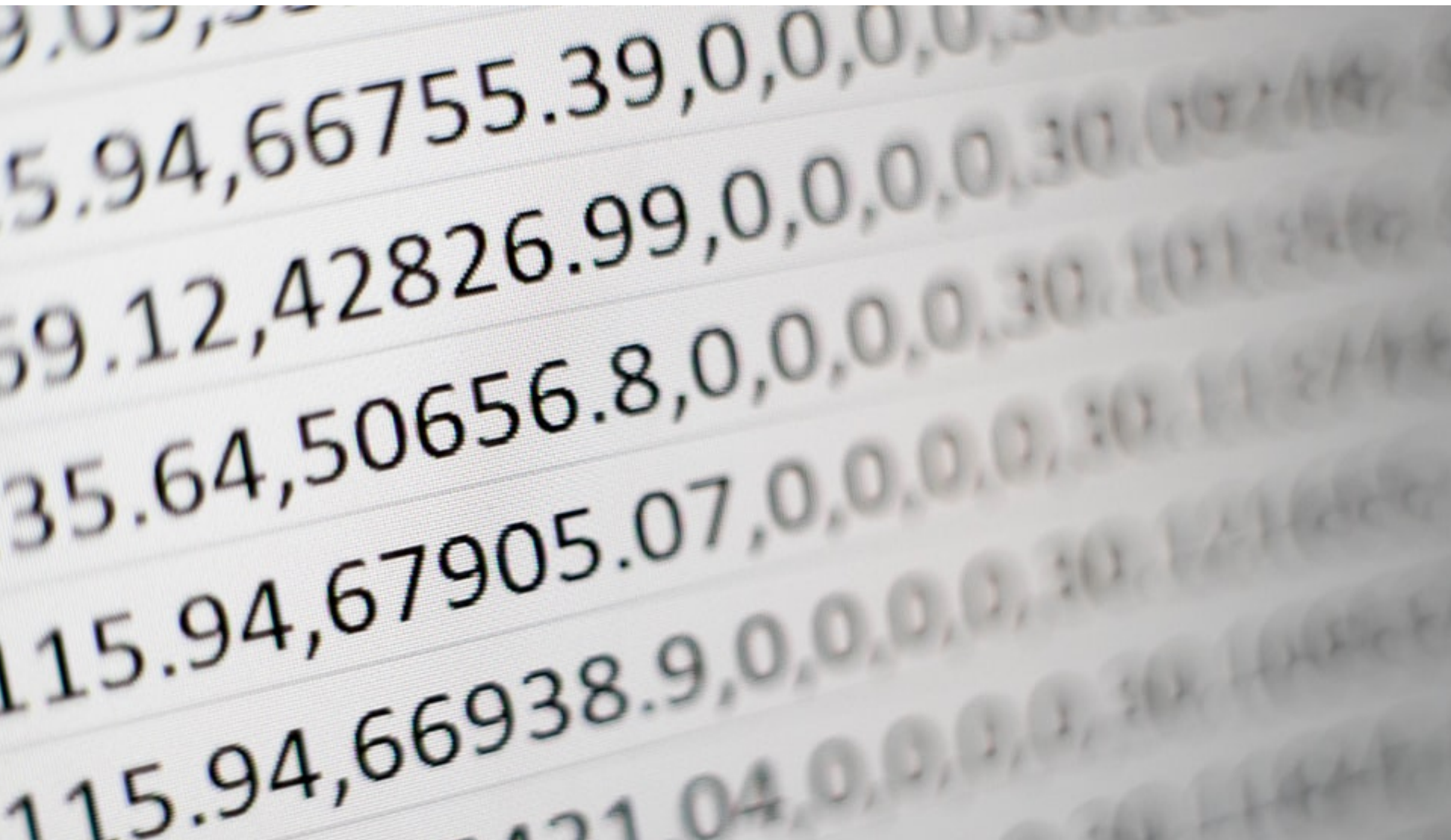
- High level API
- Robust and effective
- Production ready

- .fit .transform methods
- A host of classes for data processing
- Cross-validation

Let's dive into data pipeline

- Start by dividing the input data into:
 - numeric variables
 - dates
 - categorical :
 - . Ordinal
 - . Low cardinality
 - . High cardinality

Numeric data



5.94,66755.39,0,0,0,0,30
59.12,42826.99,0,0,0,0,30
35.64,50656.8,0,0,0,0,30
115.94,67905.07,0,0,0,0,30
115.94,66938.9,0,0,0,0,30
121.04,0,0,0,0,30

Numeric variables

- Trash **low variance variables**
- Deal with **missing values**: just input the median, but don't forget to create missing indicators to catch any information in missingness not at random
- Strive to make your data more **gaussian-like**
- Catch **outliers** (or be confident that your BatchNormalization layer will)
- **Normalize** all the values before feeding them

Tricks and tips of the trade

- Feature selector that removes all low-variance features.

```
from sklearn.feature_selection import VarianceThreshold
```

```
X = [[0, 2, 0, 3], [0, 1, 4, 3], [0, 1, 1, 3]]  
selector = VarianceThreshold(threshold=0.0)  
selector.fit_transform(X)
```

- Imputation of missing values.

```
import numpy as np  
from sklearn.impute import SimpleImputer
```

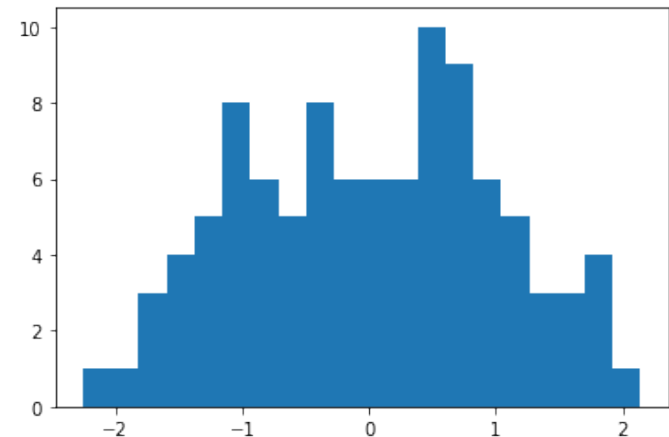
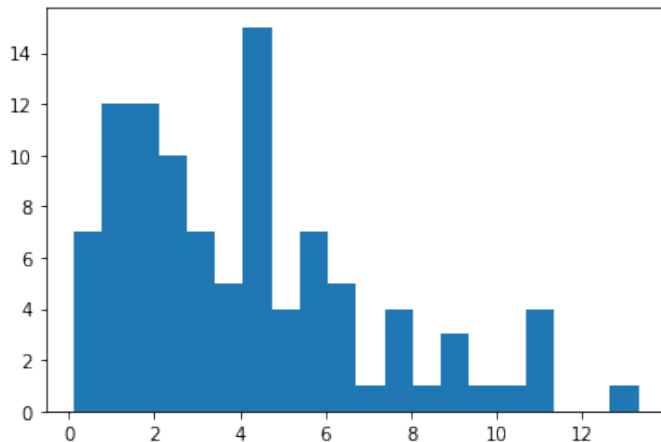
```
X = [[7, 2, 3], [4, np.nan, 6], [10, 5, 9]]  
imp_mean = SimpleImputer(missing_values=np.nan, strategy='median')  
imp_mean.fit_transform(X)
```


Tricks and tips of the trade

- Apply a power transformation to make data more Gaussian-like.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import PowerTransformer
```

```
rng = np.random.RandomState(0)
X = np.sort(rng.chisquare(4, 100), axis=0).reshape(-1, 1)
pt = PowerTransformer(method='yeo-johnson', standardize=True)
Xt = pt.fit_transform(X)
```

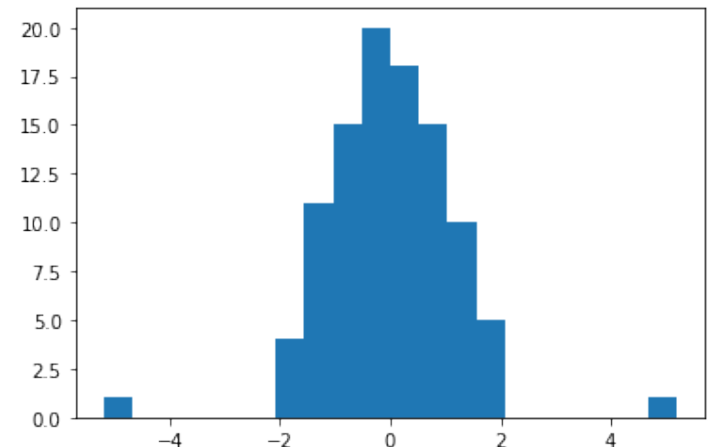
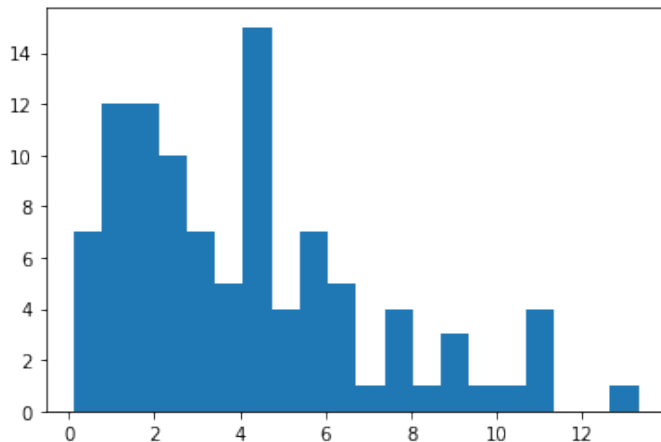


Tricks and tips of the trade

- Transform features using quantiles information spreading out the most frequent values and reducing the impact of outliers.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import QuantileTransformer
```

```
rng = np.random.RandomState(0)
X = np.sort(rng.chisquare(4, 100), axis=0).reshape(-1, 1)
qt = QuantileTransformer(n_quantiles=10, output_distribution='normal', random_state=0)
Xt = qt.fit_transform(X)
```



Tricks and tips of the trade

- Transforms by removing the mean and scaling to unit variance

```
import numpy as np
from sklearn.preprocessing import StandardScaler
```

```
X = [[ 1., -2., 2.], [-2., 1., 3.], [ 4., 1., -2.]]
sc= StandardScaler()
sc.fit_transform(X)
```

- This Scaler removes the median and scales the data according to the Interquartile Range (IQR). The IQR is the range between the 1st quartile (25th quantile) and the 3rd quartile (75th quantile).

```
import numpy as np
from sklearn.preprocessing import RobustScaler
```

```
X = [[ 1., -2., 2.], [-2., 1., 3.], [ 4., 1., -2.]]
rsc= RobustScaler()
rsc.fit_transform(X)
```

Dates



Dates

- Parse dates and split them into: **year, month, week, day of the year, day of the month, day of the week**
- Mark **special dates** or periods, such as **holidays, festivities, relevant events**
- For every split, prepare one hot encoding and **cyclic continuous transformations** (\sin / \cos)

Tricks and tips of the trade

- Using the type *datetime64*, you can easily extract the time parts

```
import pandas as pd
df = pd.DataFrame({'col1': ['02/03/2017', '02/04/2017', '02/05/2017']})
df['col1'] = df['col1'].astype('datetime64[ns]')

df.col1.dt.year, df.col1.dt.month, df.col1.dt.weekofyear,\
df.col1.dt.day, df.col1.dt.dayofyear, df.col1.dt.dayofweek
```

- Using sine and cosine transformations, create periodic features

```
import pandas as pd
import numpy as np
df = pd.DataFrame({'col1': ['02/03/2017', '02/04/2017', '02/05/2017']})
df['col1'] = df['col1'].astype('datetime64[ns]')

cycle = 7
df['weekday_sin'] = np.sin(2 * np.pi * df['col1'].dt.dayofweek / cycle)
df['weekday_cos'] = np.cos(2 * np.pi * df['col1'].dt.dayofweek / cycle)
```


Categorical data



Categorical ordinal

- For every ordinal, just let the variable as it is and one hot encoded it (you will catch both linear and non-linear ordinal effects)

```
from sklearn.preprocessing import OneHotEncoder
import pandas as pd

df = pd.DataFrame({'col1': [1, 2, 3]})
ohe = OneHotEncoder(categories='auto', sparse=False)
ohe.fit_transform(df[['col1']])
```


Low cardinality

- If a categorical variable has less than 2-255 values, just one-hot-encode.

```
from sklearn.preprocessing import OneHotEncoder
import pandas as pd
```

```
df = pd.DataFrame({'col1': ['red color', 'blue color', 'green color']})
ohe = OneHotEncoder(categories='auto', sparse=False)
ohe.fit_transform(df[['col1']])
```

High cardinality

- If a categorical variable has many different levels, convert it into numeric labels and use an embedding layer (Guo, Cheng, and Felix Berkhahn. "Entity embeddings of categorical variables." *arXiv preprint arXiv:1604.06737*, 2016).

```
from sklearn.preprocessing import LabelEncoder
import pandas as pd
```

```
df = pd.DataFrame({'col1': ['red color', 'blue color', 'green color']})
le = LabelEncoder()
le.fit_transform(df['col1'])
dictionary_length = len(le.classes_)
```

Embedding layer

- **An embedding layers is just a matrix of weights**, converting the input size (a numbered dictionary) to a lower dimensionality output (the embedding size)
- It is a **weighted linear combination** whose weights are optimized to prediction
- The resulting embedding is **exportable** and **reusable** for other problems

How an embedding layer works

Pre-processing

Text on variable: ID_8080

Label Encoding: 3 / 24

One Hot Encoding vector (size = 24,)

[0 0 1 0]

Embeddings
tf.keras.layers.
Embedding(
input_dim,
output_dim)

Weights matrix (size input x output, i.e. 24x4)

[[0.48 0.997 0.706 0.475]

...

[0.794 0.922 0.086 0.921]]

Output vector (size = 4,)

[0.283, 0.34 , 0.789, 0.867]

Backpropagation

Fully connected

Dense layers

Target

Storing away embeddings

- **Pinterest:** “Applying deep learning to Related Pins”
(Pin2vec : <https://medium.com/the-graph/applying-deep-learning-to-related-pins-a6fee3c92f5e>)

- **Instacart:** “Deep Learning with Emojis (not Math)”
(embeddings for grocery items: <https://tech.instacart.com/deep-learning-with-emojis-not-math-660ba1ad6cdc>)

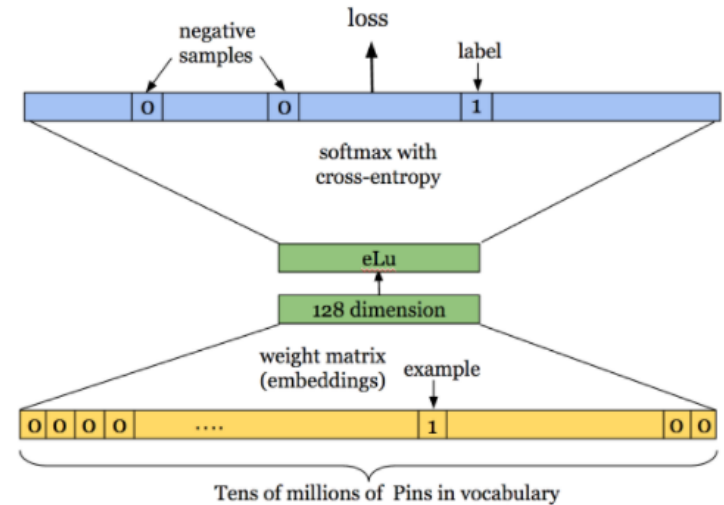


Figure 4. Feedforward neural network architecture of training Pin2Vec.

```
layer_no = 10
emb =
model.layers[layer_no].get_weights()[0]

np.save(file="emb.sav", arr=emb)
```

Dirty low and high cardinality

- If a categorical variable has dirty/fuzzy values, convert it into a sequence of shingles and feed it to an embedding layer
- Cerda, Patricio, Gaël Varoquaux, and Balázs Kégl. "Similarity encoding for learning with dirty categorical variables." *Machine Learning* 107.8-10 (2018): 1477-1494.

From abstract: “We draw practical recommendations for encoding dirty categories: 3-gram similarity appears to be a good choice to capture morphological resemblance.”

n -grams for approximate matching

- Pfizer Inc. -> ['Pfi', 'fiz', 'ize', 'zer', 'er ', 'r l', 'ln', 'Inc', 'nc.']
- Pfizer LCC -> ['Pfi', 'fiz', 'ize', 'zer', 'er ', 'r L', 'LC', 'LCC']
- By converting a string to a set of n -grams, it can be embedded in a **vector space**, thus allowing the sequence to be compared to other sequences in an efficient manner.

Generating shingles

```
from sklearn.feature_extraction.text import CountVectorizer
import pandas as pd
```

```
df = pd.DataFrame({'col1': ['Pfizer Inc.', 'Pfizer Pharmaceuticals LLC', 'Pfizer International LLC',
                             'Pfizer Limited', 'Pfizer Corporation Hong Kong Limited', 'Pfizer Pharmaceuticals Korea Limited']})
```

```
cvect = CountVectorizer(ngram_range=(3, 3), analyzer='char')
cvect.fit(df['col1'])
```

```
def nonzero_byrow(X):
    sequence = list()
    for row in range(X.shape[0]):
        _, nonzero = np.nonzero(X[row, :])
        sequence.append(nonzero)
    return sequence
```

```
nonzero_byrow(cvect.transform(df['col1']))
```


Cross validation

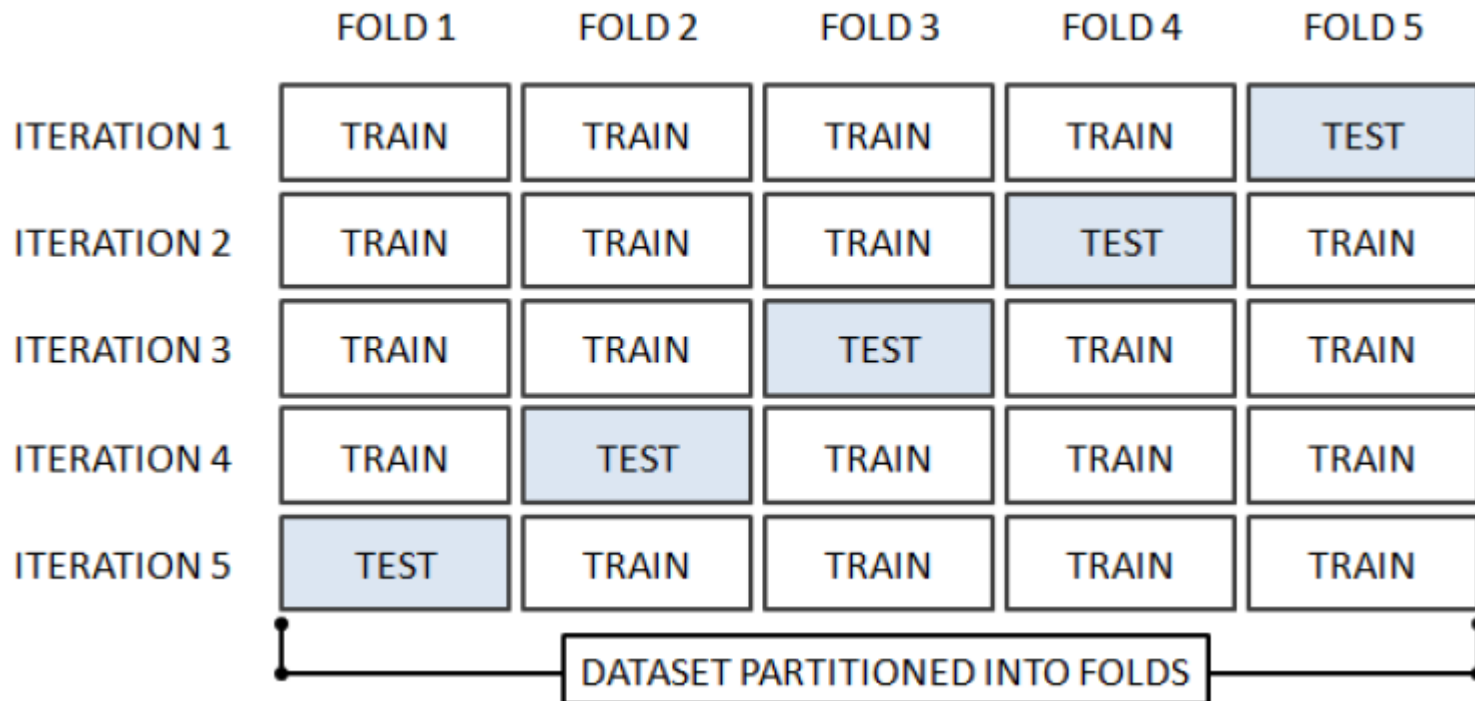


Cross validation is the key

- Unless you have an incredible amount of examples, don't bother building a generator
- If you do, keep it simple, allowing to segregate (filter), training from testing examples
- Instead, have care to create a cross-validation loop to check the performances of an overly parameterized model (params >> examples)

Cross validation primer

- It splits your data into a number k of *folds* (*portions of your data*) of equal size. Then, each fold is held out in turn as a test set and the others are used for training. Each iteration generates a representative error estimate.



Cross validation primer

```
import numpy as np
from sklearn.model_selection import KFold

X = ["a", "b", "c", "d"]
kf = KFold(n_splits=2)
for train, test in kf.split(X):
    print("%s %s" % (train, test))
```

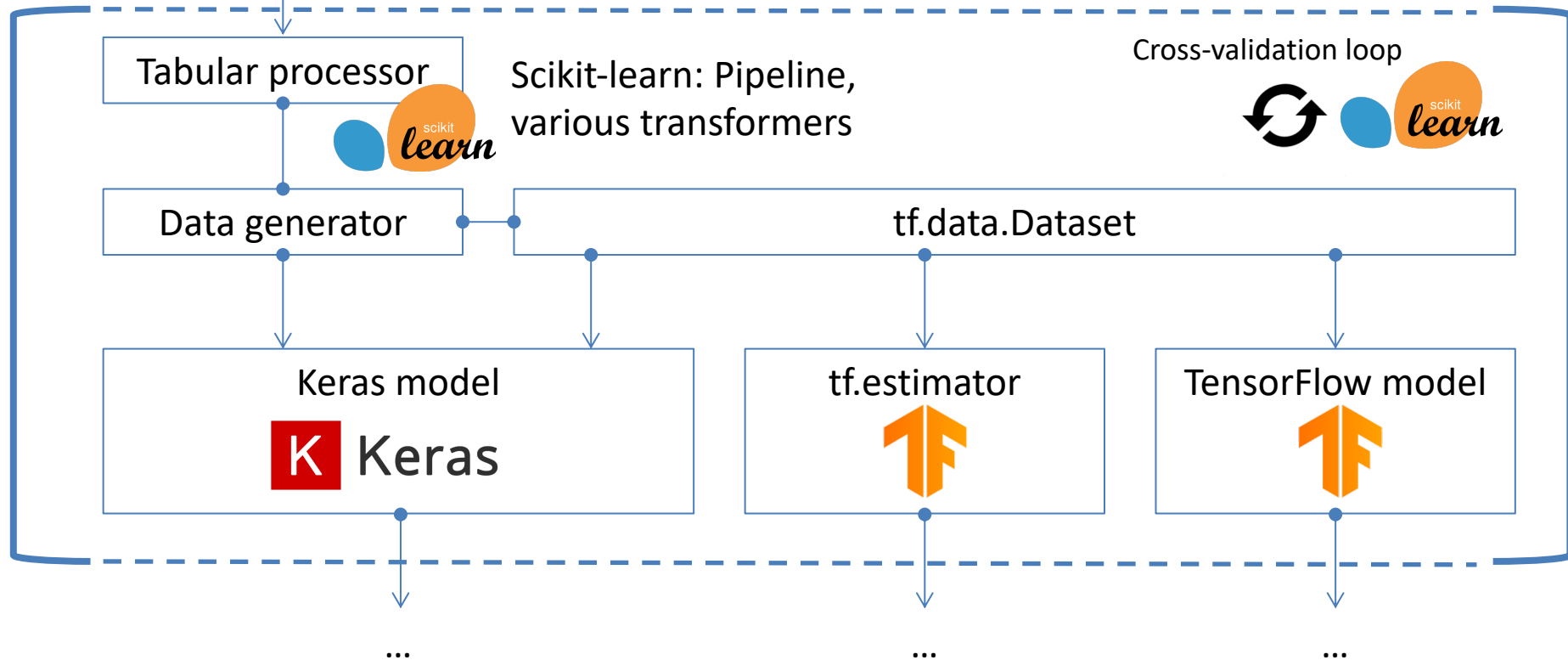
- **KFold** divides all the samples in groups of samples, called folds of equal sizes (if possible)
- **StratifiedKFold** is a variation of k-fold which returns stratified folds: each set contains approximately the same percentage of samples of each target class as the complete set.
- **GroupKFold** is a variation of k-fold which ensures that the same group is not represented in both testing and training sets.
- **TimeSeriesSplit** is a variation of k-fold which returns first folds as train set and the th fold as test set.

Wrapping up for TF



Machine Learning Pipeline

X, y as pandas DataFrame



https://github.com/lmassaron/deep_learning_for_tabular_data

Tabular processor

```
class TabularTransformer(BaseEstimator, TransformerMixin):  
  
    def __init__(self, numeric=list(), ordinal=list(), lowcat=list(), highcat=list()):  
        self.numeric = numeric  
        self.ordinal = ordinal  
        self.lowcat = lowcat  
        self.highcat = highcat  
        ...  
  
    def fit(self, X, y=None, **fit_params):  
        ...  
  
    def transform(self, X, y=None, **fit_params):  
        ...  
  
    def fit_transform(self, X, y=None, **fit_params):  
        self.fit(X, y, **fit_params)  
        return self.transform(X)
```

Data Generator

```
class DataGenerator(tf.keras.utils.Sequence):

    def __init__(self, X, y, tabular_transformer, batch_size=32,
                 shuffle=False, dict_output=False):
        self.tbtf = tabular_transformer
        ...

    def __iter__(self):
        ...

    def __next__(self):
        ...

    def __call__(self):
        ...

    def __data_generation(self, selection):
        return self.tbtf.transform(self.X.iloc[selection, :]), self.y[selection]

    def __getitem__(self, index):
        indexes = self.indexes[index*self.batch_size:(index+1)*self.batch_size]
        samples, labels = self.__data_generation(indexes)
        return samples, labels
```


tf.data

```
import tensorflow as tf

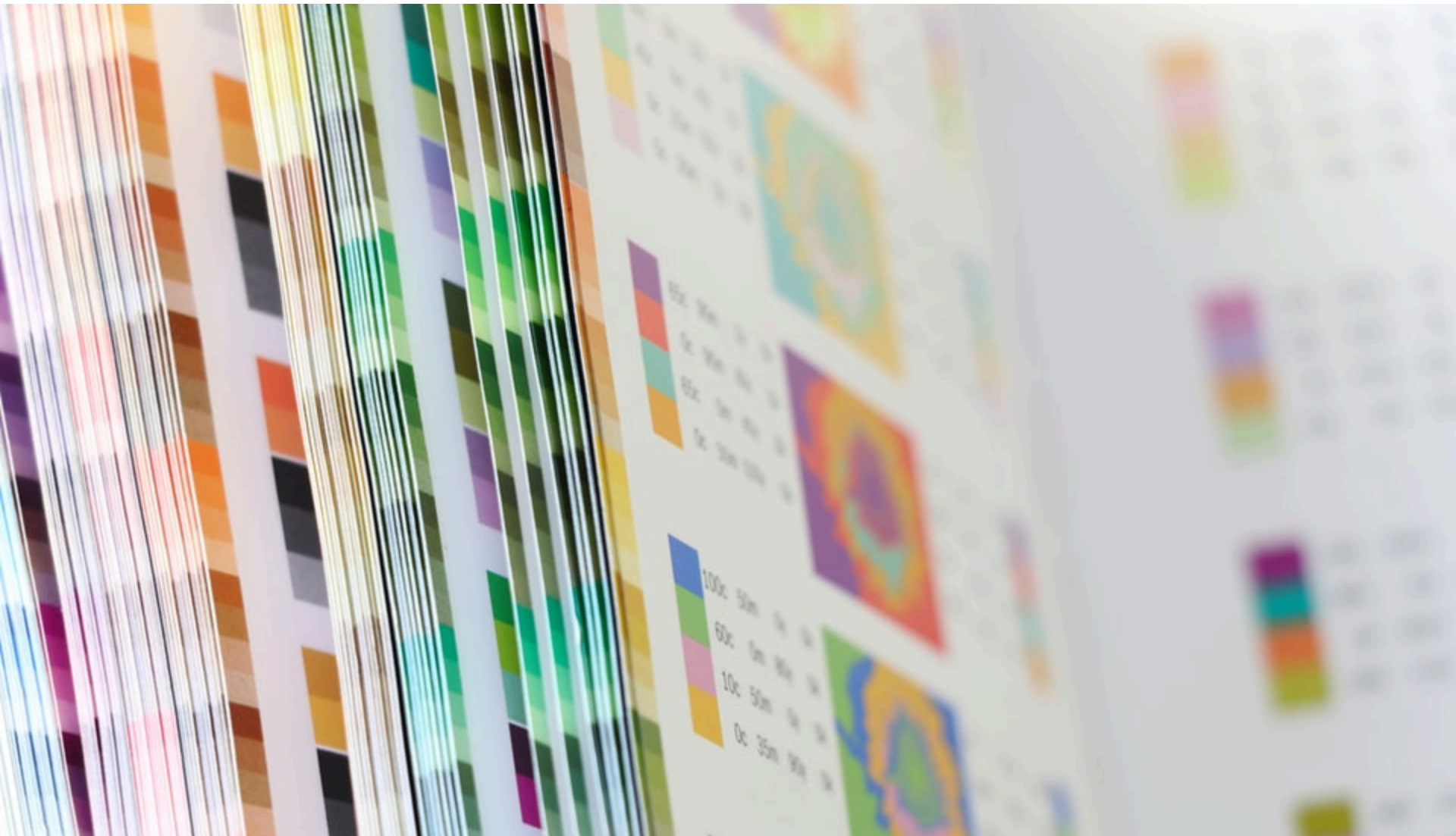
tb = TabularTransformer(numeric=num_cols, lowcat=lcat_cols, highcat=hcat_cols)

tb.fit(X)
sizes=tb.shape(X)
print(f"output dimensionality: {sizes}")

training_generator = DataGenerator(X, y, tabular_transformer=tb,
                                   batch_size=10, shuffle=True, dict_output=True)

dataset = tf.data.Dataset.from_generator(training_generator,
                                         output_types=({"input_0": tf.float32, "input_1": tf.int32}, tf.int32))
```

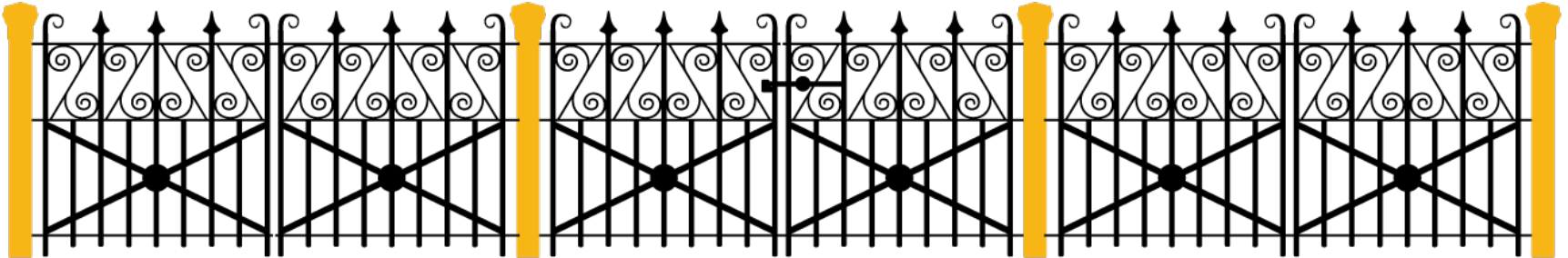
Neural architectures



Define the right architecture

- Start small, try first a shallow layer with linear activations
- Then try adding non linearity (relu, leaky relu, elu, selu, gelu)
- Try stacking more layers (2 to 3 at first, then more)
- Filter inputs using BatchNormalization
- Use dropout to control overfitting (but remember it doesn't get along with batch normalization) or, if you are using a shallow architecture, L2 regularization.
- Embed the logic you devise into a parametric function

An example



- Based on the Kaggle competition, “*Amazon.com - Employee Access Challenge*”, a competition notable for the high cardinality variables involved, the example compares the development and performances of a GBM model built with the CatBoost algorithm and a deep neural network for tabular data.

<https://www.kaggle.com/lucamassaron/deep-learning-for-tabular-data>

A peek at the used DNN

```
def tabular_dnn(numeric_variables, categorical_variables, categorical_counts,
                feature_selection_dropout=0.2, categorical_dropout=0.1,
                first_dense = 256, second_dense = 256, dense_dropout = 0.2, ):

    numerical_inputs = Input(shape=(len(numeric_variables)),)
    numerical_normalization = BatchNormalization()(numerical_inputs)
    numerical_feature_selection = Dropout(feature_selection_dropout)(numerical_normalization)

    categorical_inputs = []
    categorical_embeddings = []
    for category in categorical_variables:
        categorical_inputs.append(Input(shape=[1], name=category))
        category_counts = categorical_counts[category]
        categorical_embeddings.append(
            Embedding(category_counts+1,
                      int(np.log1p(category_counts)+1),
                      name = category + "_embed")(categorical_inputs[-1]))

    categorical_logits = Concatenate()(
        [Flatten()(SpatialDropout1D(categorical_dropout)(cat_emb))
         for cat_emb in categorical_embeddings])
```

A peek at the used DNN

```
x = concatenate([numerical_feature_selection, categorical_logits])
x = Dense(first_dense, activation='gelu')(x)
x = Dropout(dense_dropout)(x)
x = Dense(second_dense, activation='gelu')(x)
x = Dropout(dense_dropout)(x)
output = Dense(1, activation="sigmoid")(x)
model = Model([numerical_inputs] + categorical_inputs, output)

return model
```

- The deep learning model presented in the notebook is compared to CatBoost, a gradient boosting solution which is considered as the state of the art for tabular data with categorical variables.
- Castboost can achieve a test score of **0.876 Roc Auc score**, the DNN model **0.863**, when both blended together, **0.882.**, demonstrating how a blend between a GBM and a DNN achieves best results.

Pro-tips (but there's no free lunch)

- Try different activations: start first with ReLU and Leaky and then try more specific functions such as SeLU, GeLU (Gaussian Error Linear Unit), Mish. Just make them custom objects using `get_custom_objects().update({})`
- Try cosine similarity loss in place of log-loss: for some problems it resulted in better results
- Try ensembling more DNNs by sampling rows (sub-sampling or bootstrapping) and columns
- Try augmenting the examples available by adding random noise, using SMOTE or Mixup augmentation

Conclusion



Wrap up

- DNN for tabular data are an **alternative solution** for modelling a response given tabular data
- They require an ad-hoc architecture and a well crafted pipeline, so they are not effortless, but you can get far integrating **TensorFlow** and **Scikit-learn** in your code
- **Entity Embeddings** of categorical data can be of great use when dealing with high dimensional data relevant to the problem
- Most of the time they can be competitive with GBM solutions, and often they complete them (**blending can help reaching better and more stable models**)

Thank you!! Q&A

Luca Massaron

Google Developer Expert



https://developers.google.com/community/experts/directory/profile/profile-luca_massaron

 <https://www.linkedin.com/in/lmassaron/>

 <https://twitter.com/lucamassaron>

 <https://github.com/lmassaron>

 <https://www.amazon.com/Luca-Massaron/e/B00RW7GV02>