Open in Colab

T81-558: Applications of Deep Neural Networks

Module 2: Python for Machine Learning

- Instructor: Jeff Heaton, McKelvey School of Engineering, Washington University in St. Louis
- For more information visit the class website.

Module 2 Material

Main video lecture:

- Part 2.1: Introduction to Pandas [Video] [Notebook]
- **Part 2.2: Categorical Values** [Video] [Notebook]
- Part 2.3: Grouping, Sorting, and Shuffling in Python Pandas [Video] [Notebook]
- Part 2.4: Using Apply and Map in Pandas for Keras [Video] [Notebook]
- Part 2.5: Feature Engineering in Pandas for Deep Learning in Keras [Video] [Notebook]

Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

In [1]:
```
try:
    %tensorflow_version 2.x
    COLAB = True
    print("Note: using Google CoLab")
except:
    print("Note: not using Google CoLab")
    COLAB = False
```

```
Note: not using Google CoLab
```

Part 2.2: Categorical and Continuous Values

Neural networks require their input to be a fixed number of columns. This input format is very similar to spreadsheet data; it must be entirely numeric. It is essential to represent the data so that the neural network can train from it. Before we look at specific ways to preprocess data, it is important to consider four basic types of data, as defined by [Cite:stevens1946theory]. Statisticians commonly refer to as the levels of measure:

- Character Data (strings)
  - **Nominal** - Individual discrete items, no order. For example, color, zip code, and shape.
  - **Ordinal** - Individual distinct items have an implied order. For example, grade level, job title, Starbucks(tm) coffee size (tall, vente, grande)
- Numeric Data
  - **Interval** - Numeric values, no defined start. For example, temperature. You would never say, "yesterday was twice as hot as today."
  - **Ratio** - Numeric values, clearly defined start. For example, speed. You could say, "The first car is going twice as fast as the second."

Encoding Continuous Values

One common transformation is to normalize the inputs. It is sometimes valuable to normalize numeric inputs in a standard form so that the program can easily compare these two values. Consider if a friend told you that he received a 10-dollar discount. Is this a good deal? Maybe. But the cost is not normalized. If your friend purchased a car, the discount is not that good. If your friend bought lunch, this is an excellent discount!

Percentages are a prevalent form of normalization. If your friend tells you they got 10% off, we know that this is a better discount than 5%. It does not matter how much the purchase price was. One widespread machine learning normalization is the Z-Score:

$$z = \frac{x - \mu}{\sigma}$$

To calculate the Z-Score, you also need to calculate the mean(μ or $\bar{x}$) and the standard deviation (σ). You can calculate the mean with this equation:

$$\mu = \bar{x} = \frac{x_1 + x_2 + \cdots + x_n}{n}$$

The standard deviation is calculated as follows:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

The following Python code replaces the mpg with a z-score. Cars with average MPG will be near zero, above zero is above average, and below zero is below average. Z-Scores more that 3 above or below are very rare; these are outliers.

In [2]:
```
import os
import pandas as pd
from scipy.stats import zscore

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA','?'])

pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)

df['mpg'] = zscore(df['mpg'])
display(df)
```

	mpg	cylinders	displacement	...	year	origin	name
0	-0.706439	8	307.0	...	70	1	chevrolet chevelle malibu
1	-1.090751	8	350.0	...	70	1	buick skylark 320
...	...	...	...	...	...	...	...
396	0.574601	4	120.0	...	82	1	ford ranger
397	0.958913	4	119.0	...	82	1	chevy s-10

398 rows × 9 columns

Encoding Categorical Values as Dummies

The traditional means of encoding categorical values is to make them dummy variables. This technique is also called one-hot-encoding. Consider the following data set.

In [3]:
```
import pandas as pd

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/jh-simple-dataset.csv",
    na_values=['NA','?'])

pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)
```

```
display(df)
```

	id	job	area	...	retail_dense	crime	product
0	1	vv	c	...	0.492126	0.071100	b
1	2	kd	c	...	0.342520	0.400809	c
...	...	...	...	...	...	...	...
1998	1999	qp	c	...	0.598425	0.117803	c
1999	2000	pe	c	...	0.539370	0.451973	c

2000 rows × 14 columns

The *area* column is not numeric, so you must encode it with one-hot encoding. We display the number of areas and individual values. There are just four values in the *area* categorical variable in this case.

In [4]:
```
areas = list(df['area'].unique())
print(f'Number of areas: {len(areas)}')
print(f'Areas: {areas}')
```

```
Number of areas: 4
Areas: ['c', 'd', 'a', 'b']
```

There are four unique values in the *area* column. To encode these dummy variables, we would use four columns, each representing one of the areas. For each row, one column would have a value of one, the rest zeros. For this reason, this type of encoding is sometimes called one-hot encoding. The following code shows how you might encode the values "a" through "d." The value A becomes [1,0,0,0] and the value B becomes [0,1,0,0].

In [5]:
```
dummies = pd.get_dummies(['a','b','c','d'],prefix='area')
print(dummies)
```

```
   area_a  area_b  area_c  area_d
0       1       0       0       0
1       0       1       0       0
2       0       0       1       0
3       0       0       0       1
```

We can now encode the actual column.

In [6]:
```
dummies = pd.get_dummies(df['area'],prefix='area')
print(dummies[0:10]) # Just show the first 10
```

```
   area_a  area_b  area_c  area_d
0       0       0       1       0
1       0       0       1       0
..    ...     ...     ...     ...
8       0       0       1       0
9       1       0       0       0

[10 rows x 4 columns]
```

For the new dummy/one hot encoded values to be of any use, they must be merged back into the data set.

In [7]:
```
df = pd.concat([df,dummies],axis=1)
```

To encode the *area* column, we use the following code. Note that it is necessary to merge these dummies back into the data frame.

In [8]:
```
pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 10)

display(df[['id','job','area','income','area_a',
                  'area_b','area_c','area_d']])
```

	id	job	area	income	area_a	area_b	area_c	area_d
0	1	vv	c	50876.0	0	0	1	0
1	2	kd	c	60369.0	0	0	1	0
2	3	pe	c	55126.0	0	0	1	0
3	4	11	c	51690.0	0	0	1	0
4	5	kl	d	28347.0	0	0	0	1
...	...	...	...	...	...	...	...	...
1995	1996	vv	c	51017.0	0	0	1	0
1996	1997	kl	d	26576.0	0	0	0	1
1997	1998	kl	d	28595.0	0	0	0	1
1998	1999	qp	c	67949.0	0	0	1	0
1999	2000	pe	c	61467.0	0	0	1	0

2000 rows × 8 columns

Usually, you will remove the original column *area* because the goal is to get the data frame to be entirely numeric for the neural network.

In [9]:
```
pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 5)

df.drop('area', axis=1, inplace=True)
```

```
display(df[['id','job','income','area_a',
                'area_b','area_c','area_d']])
```

	id	job	income	area_a	area_b	area_c	area_d
0	1	vv	50876.0	0	0	1	0
1	2	kd	60369.0	0	0	1	0
...	...	...	...	...	...	...	...
1998	1999	qp	67949.0	0	0	1	0
1999	2000	pe	61467.0	0	0	1	0

2000 rows × 7 columns

Removing the First Level

The **pd.concat** function also includes a parameter named *drop_first*, which specifies whether to get k-1 dummies out of k categorical levels by removing the first level. Why would you want to remove the first level, in this case, *area_a*? This technique provides a more efficient encoding by using the ordinarily unused encoding of [0,0,0]. We encode the *area* to just three columns and map the categorical value of *a* to [0,0,0]. The following code demonstrates this technique.

In [10]:
```
import pandas as pd

dummies = pd.get_dummies(['a','b','c','d'],prefix='area', drop_first=True)
print(dummies)
```

```
   area_b  area_c  area_d
0       0       0       0
1       1       0       0
2       0       1       0
3       0       0       1
```

As you can see from the above data, the *area_a* column is missing, as it **get_dummies** replaced it by the encoding of [0,0,0]. The following code shows how to apply this technique to a dataframe.

In [11]:
```
import pandas as pd

# Read the dataset
df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/jh-simple-dataset.csv",
    na_values=['NA','?'])

# encode the area column as dummy variables
dummies = pd.get_dummies(df['area'], drop_first=True, prefix='area')
df = pd.concat([df,dummies],axis=1)
df.drop('area', axis=1, inplace=True)
```

```
# display the encoded dataframe
pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 10)

display(df[['id','job','income',
                  'area_b','area_c','area_d']])
```

	id	job	income	area_b	area_c	area_d
0	1	vv	50876.0	0	1	0
1	2	kd	60369.0	0	1	0
2	3	pe	55126.0	0	1	0
3	4	11	51690.0	0	1	0
4	5	kl	28347.0	0	0	1
...	...	...	...	...	...	...
1995	1996	vv	51017.0	0	1	0
1996	1997	kl	26576.0	0	0	1
1997	1998	kl	28595.0	0	0	1
1998	1999	qp	67949.0	0	1	0
1999	2000	pe	61467.0	0	1	0

2000 rows × 6 columns

Target Encoding for Categoricals

Target encoding is a popular technique for Kaggle competitions. Target encoding can sometimes increase the predictive power of a machine learning model. However, it also dramatically increases the risk of overfitting. Because of this risk, you must take care of using this method.

Generally, target encoding can only be used on a categorical feature when the output of the machine learning model is numeric (regression).

The concept of target encoding is straightforward. For each category, we calculate the average target value for that category. Then to encode, we substitute the percent corresponding to the category that the categorical value has. Unlike dummy variables, where you have a column for each category with target encoding, the program only needs a single column. In this way, target coding is more efficient than dummy variables.

In [13]:
```
# Create a small sample dataset
import pandas as pd
import numpy as np
```

```
np.random.seed(43)
df = pd.DataFrame({
    'cont_9': np.random.rand(10)*100,
    'cat_0': ['dog'] * 5 + ['cat'] * 5,
    'cat_1': ['wolf'] * 9 + ['tiger'] * 1,
    'y': [1, 0, 1, 1, 1, 1, 0, 0, 0, 0]
})

pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 0)
display(df)
```

	cont_9	cat_0	cat_1	y
0	11.505457	dog	wolf	1
1	60.906654	dog	wolf	0
2	13.339096	dog	wolf	1
3	24.058962	dog	wolf	1
4	32.713906	dog	wolf	1
5	85.913749	cat	wolf	1
6	66.609021	cat	wolf	0
7	54.116221	cat	wolf	0
8	2.901382	cat	wolf	0
9	73.374830	cat	tiger	0

We want to change them to a number rather than creating dummy variables for "dog" and "cat," we would like to change them to a number. We could use 0 for a cat and 1 for a dog. However, we can encode more information than just that. The simple 0 or 1 would also only work for one animal. Consider what the mean target value is for cat and dog.

In [14]:
```
means0 = df.groupby('cat_0')['y'].mean().to_dict()
means0
```

Out[14]:
```
{'cat': 0.2, 'dog': 0.8}
```

The danger is that we are now using the target value (y) for training. This technique will potentially lead to overfitting. The possibility of overfitting is even greater if a small number of a particular category. To prevent this from happening, we use a weighting factor. The stronger the weight, the more categories with fewer values will tend towards the overall average of y. You can perform this calculation as follows.

In [15]:
```
df['y'].mean()
```

Out[15]: 0.5

You can implement target encoding as follows. For more information on Target Encoding, refer to the article "Target Encoding Done the Right Way", that I based this code upon.

In [16]:
```
def calc_smooth_mean(df1, df2, cat_name, target, weight):
    # Compute the global mean
    mean = df[target].mean()

    # Compute the number of values and the mean of each group
    agg = df.groupby(cat_name)[target].agg(['count', 'mean'])
    counts = agg['count']
    means = agg['mean']

    # Compute the "smoothed" means
    smooth = (counts * means + weight * mean) / (counts + weight)

    # Replace each value by the according smoothed mean
    if df2 is None:
        return df1[cat_name].map(smooth)
    else:
        return df1[cat_name].map(smooth),df2[cat_name].map(smooth.to_dict())
```

The following code encodes these two categories.

In [17]:
```
WEIGHT = 5
df['cat_0_enc'] = calc_smooth_mean(df1=df, df2=None,
    cat_name='cat_0', target='y', weight=WEIGHT)
df['cat_1_enc'] = calc_smooth_mean(df1=df, df2=None,
    cat_name='cat_1', target='y', weight=WEIGHT)

pd.set_option('display.max_columns', 0)
pd.set_option('display.max_rows', 0)

display(df)
```

	cont_9	cat_0	cat_1	y	cat_0_enc	cat_1_enc
0	11.505457	dog	wolf	1	0.65	0.535714
1	60.906654	dog	wolf	0	0.65	0.535714
2	13.339096	dog	wolf	1	0.65	0.535714
3	24.058962	dog	wolf	1	0.65	0.535714
4	32.713906	dog	wolf	1	0.65	0.535714
5	85.913749	cat	wolf	1	0.35	0.535714
6	66.609021	cat	wolf	0	0.35	0.535714
7	54.116221	cat	wolf	0	0.35	0.535714
8	2.901382	cat	wolf	0	0.35	0.535714
9	73.374830	cat	tiger	0	0.35	0.416667

Encoding Categorical Values as Ordinal

Typically categoricals will be encoded as dummy variables. However, there might be other techniques to convert categoricals to numeric. Any time there is an order to the categoricals, a number should be used. Consider if you had a categorical that described the current education level of an individual.

- Kindergarten (0)
- First Grade (1)
- Second Grade (2)
- Third Grade (3)
- Fourth Grade (4)
- Fifth Grade (5)
- Sixth Grade (6)
- Seventh Grade (7)
- Eighth Grade (8)
- High School Freshman (9)
- High School Sophomore (10)
- High School Junior (11)
- High School Senior (12)
- College Freshman (13)
- College Sophomore (14)
- College Junior (15)
- College Senior (16)
- Graduate Student (17)
- PhD Candidate (18)
- Doctorate (19)

- Post Doctorate (20)

The above list has 21 levels and would take 21 dummy variables to encode. However, simply encoding this to dummies would lose the order information. Perhaps the most straightforward approach would be to simply number them and assign the category a single number equal to the value in the parenthesis above. However, we might be able to do even better. A graduate student is likely more than a year so you might increase one value.

High Cardinality Categorical

If there were many, perhaps thousands or tens of thousands, then one-hot encoding is no longer a good choice. We call these cases high cardinality categorical. We generally encode such values with an embedding layer, which we will discuss later when introducing natural language processing (NLP).