Open in Colab

T81-558: Applications of Deep Neural Networks

Module 4: Training for Tabular Data

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

Module 4 Material

- **Part 4.1: Encoding a Feature Vector for Keras Deep Learning** [Video] [Notebook]
- Part 4.2: Keras Multiclass Classification for Deep Neural Networks with ROC and AUC [Video] [Notebook]
- Part 4.3: Keras Regression for Deep Neural Networks with RMSE [Video] [Notebook]
- Part 4.4: Backpropagation, Nesterov Momentum, and ADAM Neural Network Training [Video] [Notebook]
- Part 4.5: Neural Network RMSE and Log Loss Error Calculation from Scratch [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 4.1: Encoding a Feature Vector for Keras Deep Learning

Neural networks can accept many types of data. We will begin with tabular data, where there are well-defined rows and columns. This data is what you would typically see in Microsoft Excel. Neural networks require numeric input. This numeric form is called a feature vector. Each input neurons receive one feature (or column) from this vector. Each row of training data typically becomes one vector. This section will see how to encode the following tabular data into a feature vector. You can see an example of tabular data below.

In [2]:
```
import pandas as pd

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

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

pd.set_option('display.max_columns', 9)
pd.set_option('display.max_rows', 5)

display(df)
```

	id	job	area	income	...	pop_dense	retail_dense	crime	product
0	1	vv	c	50876.0	...	0.885827	0.492126	0.071100	b
1	2	kd	c	60369.0	...	0.874016	0.342520	0.400809	c
...	...	...	...	...	...	...	...	...	...
1998	1999	qp	c	67949.0	...	0.909449	0.598425	0.117803	c
1999	2000	pe	c	61467.0	...	0.925197	0.539370	0.451973	c

2000 rows × 14 columns

You can make the following observations from the above data:

- The target column is the column that you seek to predict. There are several candidates here. However, we will initially use the column "product". This field specifies what product someone bought.
- There is an ID column. You should exclude his column because it contains no information useful for prediction.
- Many of these fields are numeric and might not require further processing.
- The income column does have some missing values.
- There are categorical values: job, area, and product.

To begin with, we will convert the job code into dummy variables.

In [3]:
```
pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)

dummies = pd.get_dummies(df['job'],prefix="job")
print(dummies.shape)

pd.set_option('display.max_columns', 9)
pd.set_option('display.max_rows', 10)

display(dummies)
```

```
(2000, 33)
```

	job_11	job_al	job_am	job_ax	...	job_rn	job_sa	job_vv	job_zz
0	0	0	0	0	...	0	0	1	0
1	0	0	0	0	...	0	0	0	0
2	0	0	0	0	...	0	0	0	0
3	1	0	0	0	...	0	0	0	0
4	0	0	0	0	...	0	0	0	0
...	...	...	...	...	...	...	...	...	...
1995	0	0	0	0	...	0	0	1	0
1996	0	0	0	0	...	0	0	0	0
1997	0	0	0	0	...	0	0	0	0
1998	0	0	0	0	...	0	0	0	0
1999	0	0	0	0	...	0	0	0	0

2000 rows × 33 columns

Because there are 33 different job codes, there are 33 dummy variables. We also specified a prefix because the job codes (such as "ax") are not that meaningful by themselves. Something such as "job_ax" also tells us the origin of this field.

Next, we must merge these dummies back into the main data frame. We also drop the original "job" field, as the dummies now represent it.

In [4]:
```
pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)

df = pd.concat([df,dummies],axis=1)
df.drop('job', axis=1, inplace=True)

pd.set_option('display.max_columns', 9)
pd.set_option('display.max_rows', 10)
```

```
display(df)
```

	id	area	income	aspect	...	job_rn	job_sa	job_vv	job_zz
0	1	c	50876.0	13.100000	...	0	0	1	0
1	2	c	60369.0	18.625000	...	0	0	0	0
2	3	c	55126.0	34.766667	...	0	0	0	0
3	4	c	51690.0	15.808333	...	0	0	0	0
4	5	d	28347.0	40.941667	...	0	0	0	0
...	...	...	...	...	...	...	...	...	...
1995	1996	c	51017.0	38.233333	...	0	0	1	0
1996	1997	d	26576.0	33.358333	...	0	0	0	0
1997	1998	d	28595.0	39.425000	...	0	0	0	0
1998	1999	c	67949.0	5.733333	...	0	0	0	0
1999	2000	c	61467.0	16.891667	...	0	0	0	0

2000 rows × 46 columns

We also introduce dummy variables for the area column.

In [5]:
```
pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 5)

df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

pd.set_option('display.max_columns', 9)
pd.set_option('display.max_rows', 10)
display(df)
```

	id	income	aspect	subscriptions	...	area_a	area_b	area_c	area_
0	1	50876.0	13.100000	1	...	0	0	1	
1	2	60369.0	18.625000	2	...	0	0	1	
2	3	55126.0	34.766667	1	...	0	0	1	
3	4	51690.0	15.808333	1	...	0	0	1	
4	5	28347.0	40.941667	3	...	0	0	0	
...	...	...	...	...	...	...	...	...	
1995	1996	51017.0	38.233333	1	...	0	0	1	
1996	1997	26576.0	33.358333	2	...	0	0	0	
1997	1998	28595.0	39.425000	3	...	0	0	0	
1998	1999	67949.0	5.733333	0	...	0	0	1	
1999	2000	61467.0	16.891667	0	...	0	0	1	

2000 rows × 49 columns

The last remaining transformation is to fill in missing income values.

In [6]:
```
med = df['income'].median()
df['income'] = df['income'].fillna(med)
```

There are more advanced ways of filling in missing values, but they require more analysis. The idea would be to see if another field might hint at what the income was. For example, it might be beneficial to calculate a median income for each area or job category. This technique is something to keep in mind for the class Kaggle competition.

At this point, the Pandas dataframe is ready to be converted to Numpy for neural network training. We need to know a list of the columns that will make up *x* (the predictors or inputs) and *y* (the target).

The complete list of columns is:

In [7]:
```
print(list(df.columns))
```

```
['id', 'income', 'aspect', 'subscriptions', 'dist_healthy', 'save_rate', 'di
st_unhealthy', 'age', 'pop_dense', 'retail_dense', 'crime', 'product', 'job_
11', 'job_al', 'job_am', 'job_ax', 'job_bf', 'job_by', 'job_cv', 'job_de',
'job_dz', 'job_e2', 'job_f8', 'job_gj', 'job_gv', 'job_kd', 'job_ke', 'job_k
l', 'job_kp', 'job_ks', 'job_kw', 'job_mm', 'job_nb', 'job_nn', 'job_ob', 'j
ob_pe', 'job_po', 'job_pq', 'job_pz', 'job_qp', 'job_qw', 'job_rn', 'job_s
a', 'job_vv', 'job_zz', 'area_a', 'area_b', 'area_c', 'area_d']
```

This data includes both the target and predictors. We need a list with the target removed. We also remove **id** because it is not useful for prediction.

In [8]:
```
x_columns = df.columns.drop('product').drop('id')
print(list(x_columns))
```

```
['income', 'aspect', 'subscriptions', 'dist_healthy', 'save_rate', 'dist_unh
ealthy', 'age', 'pop_dense', 'retail_dense', 'crime', 'job_11', 'job_al', 'j
ob_am', 'job_ax', 'job_bf', 'job_by', 'job_cv', 'job_de', 'job_dz', 'job_e
2', 'job_f8', 'job_gj', 'job_gv', 'job_kd', 'job_ke', 'job_kl', 'job_kp', 'j
ob_ks', 'job_kw', 'job_mm', 'job_nb', 'job_nn', 'job_ob', 'job_pe', 'job_p
o', 'job_pq', 'job_pz', 'job_qp', 'job_qw', 'job_rn', 'job_sa', 'job_vv', 'j
ob_zz', 'area_a', 'area_b', 'area_c', 'area_d']
```

Generate X and Y for a Classification Neural Network

We can now generate *x* and *y*. Note that this is how we generate y for a classification problem. Regression would not use dummies and would encode the numeric value of the target.

In [9]:
```
# Convert to numpy - Classification
x_columns = df.columns.drop('product').drop('id')
x = df[x_columns].values
dummies = pd.get_dummies(df['product']) # Classification
products = dummies.columns
y = dummies.values
```

We can display the *x* and *y* matrices.

In [10]:
```
print(x)
print(y)
```

```
[[5.08760000e+04 1.31000000e+01 1.00000000e+00 ... 0.00000000e+00
  1.00000000e+00 0.00000000e+00]
 [6.03690000e+04 1.86250000e+01 2.00000000e+00 ... 0.00000000e+00
  1.00000000e+00 0.00000000e+00]
 [5.51260000e+04 3.47666667e+01 1.00000000e+00 ... 0.00000000e+00
  1.00000000e+00 0.00000000e+00]
 ...
 [2.85950000e+04 3.94250000e+01 3.00000000e+00 ... 0.00000000e+00
  0.00000000e+00 1.00000000e+00]
 [6.79490000e+04 5.73333333e+00 0.00000000e+00 ... 0.00000000e+00
  1.00000000e+00 0.00000000e+00]
 [6.14670000e+04 1.68916667e+01 0.00000000e+00 ... 0.00000000e+00
  1.00000000e+00 0.00000000e+00]]
[[0 1 0 ... 0 0 0]
 [0 0 1 ... 0 0 0]
 [0 1 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 1 0]
 [0 0 1 ... 0 0 0]
 [0 0 1 ... 0 0 0]]
```

The x and y values are now ready for a neural network. Make sure that you construct the neural network for a classification problem. Specifically,

- Classification neural networks have an output neuron count equal to the number of classes.
- Classification neural networks should use **categorical_crossentropy** and a **softmax** activation function on the output layer.

Generate X and Y for a Regression Neural Network

The program generates the *x* values the say way for a regression neural network. However, *y* does not use dummies. Make sure to replace **income** with your actual target.

In [11]:
```
y = df['income'].values
```

Module 4 Assignment

You can find the first assignment here: assignment 4

In [ ]:
```
```