LEARN DATA SCIENCE, USE CASES

August 9, 2018

Fraud Detection Using Autoencoders in Keras with a TensorFlow Backend

David Ellison

In this tutorial, we will use a neural network called an autoencoder to detect fraudulent credit/debit card transactions on a Kaggle dataset. We will introduce the importance of the business case, introduce autoencoders, perform an exploratory data analysis, and create and then evaluate the model. The model will be presented using Keras with a TensorFlow backend using a Jupyter Notebook and generally applicable to a wide range of anomaly detection problems.

Introduction

Card Fraud as a Booming Business

Nilson reports that U.S. card fraud (credit, debt, etc) was reportedly $9 billion in 2016 and expected to increase to $12 billion by 2020. For perspective, in 2017 both PayPal's and Mastercard's revenue was only $10.8 billion each.

Fraud Detection Algorithms

Traditionally, many major banks have relied on old rules-based expert systems to catch fraud, but these systems have proved all too easy to beat; the financial services industry is relying on increasing complex fraud detection algorithms. Many in the financial services industry have updated their fraud detection to include some basic machine learning algorithms including various clustering classifiers, linear approaches, and support vector machines. The most advanced companies in the financial services industry, such as PayPal, have been pioneering more advanced artificial intelligence techniques such as deep neural networks and autoencoders. Long story short, if you want to be where the industry is going and where the jobs are, focus on more advanced fraud detection techniques. This tutorial will focus on one of those more advanced techniques, autoencoders.

Autoencoders and Why You Should Use Them

Autoencoders are a type of neural network that takes an input (e.g. image, dataset), boils that input down to core features, and reverses the process to recreate the input. Although it may sound pointless to feed in input just to get the same thing out, it is in fact very useful for a number of applications. The key here is that the autoencoder boils down (encodes) the input into some key features that it determines in an unsupervised manner. Hence the name "autoencoder" — it automatically encodes the input.

Let us take this autoencoder of a bicycle as an example. The input is some actual picture of a bicycle that is then reduced to some hidden encoding (perhaps representing components such as handlebars and two wheels) and then is able to reconstruct the original object from that encoding. Of course there will be some loss ("reconstruction error") but hopefully the parts that remain will be the essential pieces of a bicycle.

Now let us assume you fed something into this autoencoder that was a unicycle trying to pose as a bicycle. In the process of breaking down the unicycle into components intended for bicycles, the reconstructed

version of the unicycle will be really altered (i.e. suffer a high reconstruction error). It is the assumption in using autoencoders that fraud or anomalies will suffer from a detectably high reconstruction error.

Import Statements

First, let's set up the code and import all the necessary packages.

In [2]:

```
# import packages
# matplotlib inline
import pandas as pd
import numpy as np
from scipy import stats
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns
import pickle
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, precision_recall_curve
from sklearn.metrics import recall_score, classification_report, auc, roc_curve
from sklearn.metrics import precision_recall_fscore_support, f1_score
from sklearn.preprocessing import StandardScaler
from pylab import rcParams
from keras.models import Model, load_model
from keras.layers import Input, Dense
from keras.callbacks import ModelCheckpoint, TensorBoard
from keras import regularizers

#set random seed and percentage of test data
RANDOM_SEED = 314 #used to help randomly select the data points
TEST_PCT = 0.2 # 20% of the data

#set up graphic style in this case I am using the color scheme from xkcd.com
rcParams['figure.figsize'] = 14, 8.7 # Golden Mean
LABELS = ["Normal","Fraud"]
col_list = ["cerulean","scarlet"]# https://xkcd.com/color/rgb/
sns.set(style='white', font_scale=1.75, palette=sns.xkcd_palette(col_list))
```

Import and Check Data

Download the credit card fraud dataset from Kaggle and place it in the same directory as your python notebook. The data contains 284,807 European credit card transactions that occurred over two days with 492 fraudulent transactions. Everything except the time and amount has been reduced by a Principle Component Analysis (PCA) for privacy concerns.

In [3]:

```
df = pd.read_csv("creditcard.csv") #unzip and read in data downloaded to the local directory
df.head(n=5) #just to check you imported the dataset properly
```

Out[3]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.247998	0.771679
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.108300	0.005274
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	-0.009431	0.798278

The data looks like we would expect on the surface, but let's double check the shape (we are expecting 294,807 rows and 31 columns). It is a well-groomed dataset so we expect no null values.

In [4]:

```
df.shape #secondary check on the size of the dataframe
```

Out[4]:

```
(284807, 31)
```

In [5]:

```
df.isnull().values.any() #check to see if any values are null, which there are not
```

Out[5]:

```
False
```

Indeed the data seems to be cleaned and loaded as we expect. Now we want to check if we have the expected number of normal and fraudulent rows of data. We will simply pull the "Class" column and count the number of normal (0) and fraud (1) rows.

In [6]:

```
pd.value_counts(df['Class'], sort = True) #class comparison 0=Normal 1=Fraud
```

Out[6]:

```
0    284315
1       492
Name: Class, dtype: int64
```

The counts are as expected (284,315 normal transactions and 492 fraud transactions). As is typical in fraud and anomaly detection in general, this is a very unbalanced dataset.

Exploratory Data Analysis

Balance of Data Visualization

Let's get a visual confirmation of the unbalanced data in this fraud dataset.

In [7]:

```
#if you don't have an intuitive sense of how imbalanced these two classes are, let's go visual
count_classes = pd.value_counts(df['Class'], sort = True)
count_classes.plot(kind = 'bar', rot=0)
plt.xticks(range(2), LABELS)
plt.title("Frequency by observation number")
plt.xlabel("Class")
plt.ylabel("Number of Observations");
```

As you can see, the normal cases strongly outweigh the fraud cases.

Summary Statistics of the Transaction Amount Data

We will cut up the dataset into two data frames, one for normal transactions and the other for fraud.

In [8]:

```
normal_df = df[df.Class == 0] #save normal_df observations into a separate df
fraud_df = df[df.Class == 1] #do the same for frauds
```

Let's look at some summary statistics and see if there are obvious differences between fraud and normal transactions.

In [9]:

```
normal_df.Amount.describe()
```

Out[9]:

```
count 284315.000000
mean 88.291022
std 250.105092
min 0.000000
25% 5.650000
50% 22.000000
75% 77.050000
max 25691.160000
Name: Amount, dtype: float64
```

In [10]:

```
fraud_df.Amount.describe()
```

Out[10]:

```
count     492.000000
mean      122.211321
std       256.683288
min         0.000000
25%         1.000000
50%         9.250000
75%       105.890000
max      2125.870000
Name: Amount, dtype: float64
```

Although the mean is a little higher in the fraud transactions, it is certainly within a standard deviation and so is unlikely to be easy to discriminate in a highly precise manner between the classes with pure statistical methods. I could run statistical tests (e.g. t-test) to support the claim that the two samples likely come from populations with similar means and deviations. However, such statistical methods are not the focus of this article on autoencoders.

Visual Exploration of the Transaction Amount Data

We are going to get more familiar with the data and try some basic visuals. In anomaly detection datasets it is common to have the areas of interest "washed out" by abundant data. The most common method is to simply 'slice and dice' the data in a couple different ways until something interesting is found. Although this practice is common, it is **not** a scientifically sound way to explore data. There are always non-meaningful quirks to real data, so just looking until you "find something interesting" is likely going to result in you finding false positives. In other words, you find a random pattern in the current data set that will never be seen again. As a famous economist wrote, "If you torture the data long enough, it will confess."

In this dataset, I expect a lot of low-value transactions that will be generally uninteresting (buying cups of coffee, lunches, etc). This abundant data is likely to wash out the rest of the data, so I decided to look at the data in a number different $100 and $1,000 intervals. Since it would be tedious to show reader these graphs, I will only show the final graph that only visualizes the transactions above $200.

In [13]:

```
#plot of high value transactions
bins = np.linspace(200, 2500, 100)
plt.hist(normal_df.Amount, bins, alpha=1, normed=True, label='Normal')
plt.hist(fraud_df.Amount, bins, alpha=0.6, normed=True, label='Fraud')
plt.legend(loc='upper right')
plt.title("Amount by percentage of transactions (transactions \$200+)")
plt.xlabel("Transaction amount (USD)")
plt.ylabel("Percentage of transactions (%)");
plt.show()
```

Since the fraud cases are relatively few in number compared to bin size, we see the data looks predictably more variable. In the long tail, especially, we are likely observing only a single fraud transaction. It would be hard to differentiate fraud from normal transactions by transaction amount alone.

Visual Exploration of the Data by Hour

With a few exceptions, the transaction amount does not look very informative. Let's look at the time of day next.

In [14]:

```
bins = np.linspace(0, 48, 48) #48 hours
plt.hist((normal_df.Time/(60*60)), bins, alpha=1, normed=True, label='Normal')
plt.hist((fraud_df.Time/(60*60)), bins, alpha=0.6, normed=True, label='Fraud')
plt.legend(loc='upper right')
plt.title("Percentage of transactions by hour")
plt.xlabel("Transaction time as measured from first transaction in the dataset (hours)")
plt.ylabel("Percentage of transactions (%)");
#plt.hist((df.Time/(60*60)),bins)
plt.show()
```

Hour "zero" corresponds to the hour the first transaction happened and not necessarily 12-1am. Given the heavy decrease in normal transactions from hours 1 to 8 and again roughly at hours 24 to 32, I am assuming those time correspond to nighttime for this dataset. If this is true, fraud tends to occur at higher rates during the night. Statistical tests could be used to give evidence for this fact, but are not in the scope of this article. Again, however, the potential time offset between normal and fraud transactions is not enough to make a simple, precise classifier.

Next, we will explore the potential interaction between transaction amount and hour to see if any patterns emerge.

Visual Exploration of Transaction Amount vs. Hour

In [15]:

```
plt.scatter((normal_df.Time/(60*60)), normal_df.Amount, alpha=0.6, label='Normal')
plt.scatter((fraud_df.Time/(60*60)), fraud_df.Amount, alpha=0.9, label='Fraud')
plt.title("Amount of transaction by hour")
plt.xlabel("Transaction time as measured from first transaction in the dataset (hours)")
plt.ylabel('Amount (USD)')
plt.legend(loc='upper right')
plt.show()
```

Again, this is not enough to make a good classifier. For example, it would be hard to draw a line that cleanly separates fraud and normal transactions. For the experienced Data Scientists in the readership, I am excluding more advanced techniques such as the kernel trick.

Model Setup: Basic Autoencoder

Now that more simplistic methods are not proving that useful, we are justified in exploring our autoencoder to see if it does a little better.

Normalize and Scale Data

Both time and amount have very different magnitudes, which will likely result in the large magnitude value "washing out" the small magnitude value. It is therefore common to scale the data to similar magnitudes. Although there are many different scaling methods and reasons to choose one method over the other, in this case I will err towards consistency. The reader may remember that most of the data (other than 'time' and 'amount') result from the product of a PCA analysis. The PCA done on the dataset transformed it into standard-normal form. I will do the same to the 'time' and 'amount' columns.

In [16]:

```
#data = df.drop(['Time'], axis=1) #if you think the var is unimportant
df_norm = df
df_norm['Time'] = StandardScaler().fit_transform(df_norm['Time'].values.reshape(-1, 1))
df_norm['Amount'] = StandardScaler().fit_transform(df_norm['Amount'].values.reshape(-1, 1))
```

Dividing Training and Test Set

Now we split the data into training and testing sets according to the percentage and with a random seed we wrote at the beginning of the code. This should have been done before the exploratory data analysis, but for ease of explanation I delayed it until right before the model.

In [17]:

```
train_x, test_x = train_test_split(df_norm, test_size=TEST_PCT, random_state=RANDOM_SEED)
train_x = train_x[train_x.Class == 0] #where normal transactions
train_x = train_x.drop(['Class'], axis=1) #drop the class column


test_y = test_x['Class'] #save the class column for the test set
test_x = test_x.drop(['Class'], axis=1) #drop the class column

train_x = train_x.values #transform to ndarray
test_x = test_x.values
```

Just confirming the new ndarray is the expected shape.

In [18]:

```
train_x.shape
```

Out[18]:

```
(227468, 30)
```

Creating the Model

Autoencoder Layer Structure and Parameters

Below we set up the structure of the autoencoder. It has symmetric encoding and decoding layers that are "dense" (e.g. full connected). The choice of the size of these layers is relatively arbitrary and generally the coder experiments with a few different layer sizes.

Remember you are reducing the input into some form of simplified encoding and then expanding it again. The input and output dimension is the feature space (e.g. 30 columns), so the encoding layer should be smaller by an amount that I expect to represent some feature. In this case, I am encoding 30 columns into 14 dimensions so I am expecting high-level features to be represented by roughly two columns (30/14 = 2.1). Of those high-level features, I am expecting them to map to roughly seven hidden/latent features in the data.

Additionally, the epochs, batch size, learning rate, learning policy, and activation functions were all set to values empirically or for reasons that can and have repeatedly filled data science books. Explanation of the balancing of these values is far beyond this tutorial, but I would refer you to excellent texts such as Hands-On Machine Learning with Scikit-Learn & TensorFlow or Deep Learning.

In [22]:

```
nb_epoch = 100
batch_size = 128
input_dim = train_x.shape[1] #num of columns, 30
encoding_dim = 14
hidden_dim = int(encoding_dim / 2) #i.e. 7
learning_rate = 1e-7

input_layer = Input(shape=(input_dim, ))
encoder = Dense(encoding_dim, activation="tanh", activity_regularizer=regularizers.l1(learning_rate))
(input_layer)
encoder = Dense(hidden_dim, activation="relu")(encoder)
decoder = Dense(hidden_dim, activation='tanh')(encoder)
decoder = Dense(input_dim, activation='relu')(decoder)
autoencoder = Model(inputs=input_layer, outputs=decoder)
```

Model Training and Logging

Below is where we set up the actual run including checkpoints and the tensorboard.

In [23]:

```
autoencoder.compile(metrics=['accuracy'],
                    loss='mean_squared_error',
                    optimizer='adam')

cp = ModelCheckpoint(filepath="autoencoder_fraud.h5",
                               save_best_only=True,
                               verbose=0)

tb = TensorBoard(log_dir='./logs',
                 histogram_freq=0,
                 write_graph=True,
                 write_images=True)

history = autoencoder.fit(train_x, train_x,
                    epochs=nb_epoch,
                    batch_size=batch_size,
```

```
                    shuffle=True,
                    validation_data=(test_x, test_x),
                    verbose=1,
                    callbacks=[cp, tb]).history
```

```
Train on 227468 samples, validate on 56962 samples
Epoch 1/100
227468/227468 [==============================] - 7s 29us/step - loss: 0.8688 - acc: 0.4782 - val_loss:
0.8266 - val_acc: 0.5893
Epoch 2/100
227468/227468 [==============================] - 5s 20us/step - loss: 0.7767 - acc: 0.6053 - val_loss:
0.7980 - val_acc: 0.6191
Epoch 3/100
227468/227468 [==============================] - 4s 19us/step - loss: 0.7575 - acc: 0.6291 - val_loss:
0.7855 - val_acc: 0.6376
Epoch 4/100
227468/227468 [==============================] - 4s 19us/step - loss: 0.7473 - acc: 0.6395 - val_loss:
0.7781 - val_acc: 0.6412
Epoch 5/100
227468/227468 [==============================] - 4s 19us/step - loss: 0.7393 - acc: 0.6487 - val_loss:
0.7705 - val_acc: 0.6581
Epoch 6/100
227468/227468 [==============================] - 4s 19us/step - loss: 0.7319 - acc: 0.6613 - val_loss:
0.7651 - val_acc: 0.6663
Epoch 7/100
227468/227468 [==============================] - 4s 19us/step - loss: 0.7265 - acc: 0.6716 - val_loss:
0.7602 - val_acc: 0.6753
Epoch 8/100
227468/227468 [==============================] - 5s 21us/step - loss: 0.7223 - acc: 0.6777 - val_loss:
0.7569 - val_acc: 0.6803
Epoch 9/100
227468/227468 [==============================] - 4s 19us/step - loss: 0.7185 - acc: 0.6845 - val_loss:
0.7526 - val_acc: 0.6867
Epoch 10/100
227468/227468 [==============================] - 4s 19us/step - loss: 0.7148 - acc: 0.6900 - val_loss:
0.7500 - val_acc: 0.6921
Epoch 11/100
227468/227468 [==============================] - 5s 20us/step - loss: 0.7122 - acc: 0.6923 - val_loss:
0.7456 - val_acc: 0.6938
Epoch 12/100
227468/227468 [==============================] - 5s 23us/step - loss: 0.7101 - acc: 0.6939 - val_loss:
0.7441 - val_acc: 0.6903
Epoch 13/100
227468/227468 [==============================] - 6s 24us/step - loss: 0.7084 - acc: 0.6966 - val_loss:
0.7429 - val_acc: 0.6963
Epoch 14/100
227468/227468 [==============================] - 5s 20us/step - loss: 0.7073 - acc: 0.6969 - val_loss:
0.7412 - val_acc: 0.6995
Epoch 15/100
227468/227468 [==============================] - 5s 21us/step - loss: 0.7065 - acc: 0.6982 - val_loss:
0.7408 - val_acc: 0.7010
Epoch 16/100
227468/227468 [==============================] - 4s 19us/step - loss: 0.7060 - acc: 0.6987 - val_loss:
0.7406 - val_acc: 0.7029
Epoch 17/100
227468/227468 [==============================] - 5s 21us/step - loss: 0.7053 - acc: 0.6989 - val_loss:
0.7405 - val_acc: 0.6977
Epoch 18/100
227468/227468 [==============================] - 5s 23us/step - loss: 0.7049 - acc: 0.6996 - val_loss:
0.7408 - val_acc: 0.7030
Epoch 19/100
227468/227468 [==============================] - 5s 21us/step - loss: 0.7044 - acc: 0.6995 - val_loss:
0.7388 - val_acc: 0.6988
Epoch 20/100
227468/227468 [==============================] - 5s 21us/step - loss: 0.7042 - acc: 0.6997 - val_loss:
0.7389 - val_acc: 0.7003
Epoch 21/100
227468/227468 [==============================] - 6s 24us/step - loss: 0.7041 - acc: 0.7000 - val_loss:
0.7395 - val_acc: 0.7023
Epoch 22/100
227468/227468 [==============================] - 5s 24us/step - loss: 0.7037 - acc: 0.7005 - val_loss:
0.7393 - val_acc: 0.7025
Epoch 23/100
227468/227468 [==============================] - 5s 22us/step - loss: 0.7037 - acc: 0.6995 - val_loss:
0.7377 - val_acc: 0.6994
Epoch 24/100
227468/227468 [==============================] - 5s 20us/step - loss: 0.7031 - acc: 0.6998 - val_loss:
0.7384 - val_acc: 0.6959
Epoch 25/100
227468/227468 [==============================] - 6s 25us/step - loss: 0.7029 - acc: 0.6999 - val_loss:
0.7378 - val_acc: 0.7061
Epoch 26/100
227468/227468 [==============================] - 5s 24us/step - loss: 0.7030 - acc: 0.6997 - val_loss:
0.7373 - val_acc: 0.7016
Epoch 27/100
227468/227468 [==============================] - 5s 21us/step - loss: 0.7027 - acc: 0.6995 - val_loss:
0.7380 - val_acc: 0.7038
Epoch 28/100
227468/227468 [==============================] - 4s 19us/step - loss: 0.7024 - acc: 0.6998 - val_loss:
0.7368 - val_acc: 0.7028
Epoch 29/100
227468/227468 [==============================] - 4s 19us/step - loss: 0.7023 - acc: 0.6995 - val_loss:
0.7376 - val_acc: 0.7034
Epoch 30/100
```

```
227468/227468 [==============================] - 4s 20us/step - loss: 0.7021 - acc: 0.7002 - val_loss:
0.7389 - val_acc: 0.7040
Epoch 31/100
227468/227468 [==============================] - 5s 20us/step - loss: 0.7020 - acc: 0.6998 - val_loss:
0.7382 - val_acc: 0.6970
Epoch 32/100
227468/227468 [==============================] - 4s 19us/step - loss: 0.7018 - acc: 0.7000 - val_loss:
0.7384 - val_acc: 0.6972
Epoch 33/100
227468/227468 [==============================] - 4s 19us/step - loss: 0.7017 - acc: 0.7008 - val_loss:
0.7371 - val_acc: 0.7030
Epoch 34/100
227468/227468 [==============================] - 4s 19us/step - loss: 0.7014 - acc: 0.7008 - val_loss:
0.7385 - val_acc: 0.7024
Epoch 35/100
227468/227468 [==============================] - 4s 19us/step - loss: 0.7015 - acc: 0.7013 - val_loss:
0.7389 - val_acc: 0.6949
Epoch 36/100
227468/227468 [==============================] - 4s 19us/step - loss: 0.7011 - acc: 0.7011 - val_loss:
0.7354 - val_acc: 0.7014
Epoch 37/100
227468/227468 [==============================] - 4s 19us/step - loss: 0.7014 - acc: 0.7010 - val_loss:
0.7359 - val_acc: 0.7027
Epoch 38/100
227468/227468 [==============================] - 4s 19us/step - loss: 0.7009 - acc: 0.7018 - val_loss:
0.7369 - val_acc: 0.7021
Epoch 39/100
227468/227468 [==============================] - 6s 26us/step - loss: 0.7008 - acc: 0.7032 - val_loss:
0.7381 - val_acc: 0.7010
Epoch 40/100
227468/227468 [==============================] - 5s 22us/step - loss: 0.7011 - acc: 0.7024 - val_loss:
0.7383 - val_acc: 0.7015
Epoch 41/100
227468/227468 [==============================] - 6s 24us/step - loss: 0.7009 - acc: 0.7024 - val_loss:
0.7373 - val_acc: 0.6942
Epoch 42/100
227468/227468 [==============================] - 4s 19us/step - loss: 0.7005 - acc: 0.7034 - val_loss:
0.7355 - val_acc: 0.7117
Epoch 43/100
227468/227468 [==============================] - 4s 19us/step - loss: 0.7004 - acc: 0.7029 - val_loss:
0.7350 - val_acc: 0.7068
Epoch 44/100
227468/227468 [==============================] - 4s 18us/step - loss: 0.7005 - acc: 0.7030 - val_loss:
0.7353 - val_acc: 0.7078
Epoch 45/100
227468/227468 [==============================] - 4s 18us/step - loss: 0.7004 - acc: 0.7040 - val_loss:
0.7354 - val_acc: 0.7082
Epoch 46/100
227468/227468 [==============================] - 4s 17us/step - loss: 0.7002 - acc: 0.7044 - val_loss:
0.7348 - val_acc: 0.7048
Epoch 47/100
227468/227468 [==============================] - 4s 17us/step - loss: 0.7003 - acc: 0.7039 - val_loss:
0.7351 - val_acc: 0.7065
Epoch 48/100
227468/227468 [==============================] - 4s 17us/step - loss: 0.7000 - acc: 0.7045 - val_loss:
0.7346 - val_acc: 0.7062
Epoch 49/100
227468/227468 [==============================] - 4s 17us/step - loss: 0.6997 - acc: 0.7047 - val_loss:
0.7353 - val_acc: 0.7121
Epoch 50/100
227468/227468 [==============================] - 4s 16us/step - loss: 0.7000 - acc: 0.7049 - val_loss:
0.7346 - val_acc: 0.7088
Epoch 51/100
227468/227468 [==============================] - 4s 17us/step - loss: 0.6997 - acc: 0.7045 - val_loss:
0.7355 - val_acc: 0.7058
Epoch 52/100
227468/227468 [==============================] - 4s 16us/step - loss: 0.6997 - acc: 0.7043 - val_loss:
0.7353 - val_acc: 0.7114
Epoch 53/100
227468/227468 [==============================] - 4s 16us/step - loss: 0.6995 - acc: 0.7048 - val_loss:
0.7378 - val_acc: 0.7015
Epoch 54/100
227468/227468 [==============================] - 4s 16us/step - loss: 0.6995 - acc: 0.7052 - val_loss:
0.7340 - val_acc: 0.7035
Epoch 55/100
227468/227468 [==============================] - 4s 16us/step - loss: 0.6993 - acc: 0.7054 - val_loss:
0.7342 - val_acc: 0.7043
Epoch 56/100
227468/227468 [==============================] - 4s 16us/step - loss: 0.6992 - acc: 0.7056 - val_loss:
0.7346 - val_acc: 0.7092
Epoch 57/100
227468/227468 [==============================] - 4s 17us/step - loss: 0.6990 - acc: 0.7056 - val_loss:
0.7340 - val_acc: 0.7056
Epoch 58/100
227468/227468 [==============================] - 4s 16us/step - loss: 0.6991 - acc: 0.7059 - val_loss:
0.7350 - val_acc: 0.7033
Epoch 59/100
227468/227468 [==============================] - 4s 17us/step - loss: 0.6990 - acc: 0.7058 - val_loss:
0.7335 - val_acc: 0.7134
Epoch 60/100
227468/227468 [==============================] - 4s 18us/step - loss: 0.6987 - acc: 0.7056 - val_loss:
0.7328 - val_acc: 0.7079
Epoch 61/100
227468/227468 [==============================] - 4s 19us/step - loss: 0.6986 - acc: 0.7056 - val_loss:
0.7324 - val_acc: 0.7090
Epoch 62/100
227468/227468 [==============================] - 4s 19us/step - loss: 0.6985 - acc: 0.7054 - val_loss:
0.7338 - val_acc: 0.7097
```

```
Epoch 63/100
227468/227468 [==============================] - 4s 18us/step - loss: 0.6987 - acc: 0.7052 - val_loss:
0.7337 - val_acc: 0.7049
Epoch 64/100
227468/227468 [==============================] - 5s 21us/step - loss: 0.6981 - acc: 0.7051 - val_loss:
0.7357 - val_acc: 0.7065
Epoch 65/100
227468/227468 [==============================] - 5s 21us/step - loss: 0.6984 - acc: 0.7044 - val_loss:
0.7331 - val_acc: 0.6995
Epoch 66/100
227468/227468 [==============================] - 5s 22us/step - loss: 0.6982 - acc: 0.7038 - val_loss:
0.7321 - val_acc: 0.7048
Epoch 67/100
227468/227468 [==============================] - 5s 21us/step - loss: 0.6978 - acc: 0.7033 - val_loss:
0.7323 - val_acc: 0.7061
Epoch 68/100
227468/227468 [==============================] - 5s 21us/step - loss: 0.6976 - acc: 0.7035 - val_loss:
0.7324 - val_acc: 0.7038
Epoch 69/100
227468/227468 [==============================] - 4s 20us/step - loss: 0.6979 - acc: 0.7032 - val_loss:
0.7337 - val_acc: 0.7021
Epoch 70/100
227468/227468 [==============================] - 5s 20us/step - loss: 0.6976 - acc: 0.7038 - val_loss:
0.7335 - val_acc: 0.7061
Epoch 71/100
227468/227468 [==============================] - 5s 22us/step - loss: 0.6978 - acc: 0.7036 - val_loss:
0.7326 - val_acc: 0.7086
Epoch 72/100
227468/227468 [==============================] - 5s 21us/step - loss: 0.6974 - acc: 0.7044 - val_loss:
0.7332 - val_acc: 0.7080
Epoch 73/100
227468/227468 [==============================] - 5s 24us/step - loss: 0.6973 - acc: 0.7036 - val_loss:
0.7328 - val_acc: 0.7054
Epoch 74/100
227468/227468 [==============================] - 6s 25us/step - loss: 0.6974 - acc: 0.7046 - val_loss:
0.7325 - val_acc: 0.7077
Epoch 75/100
227468/227468 [==============================] - 6s 26us/step - loss: 0.6974 - acc: 0.7041 - val_loss:
0.7317 - val_acc: 0.7025
Epoch 76/100
227468/227468 [==============================] - 5s 20us/step - loss: 0.6973 - acc: 0.7040 - val_loss:
0.7336 - val_acc: 0.7054
Epoch 77/100
227468/227468 [==============================] - 4s 20us/step - loss: 0.6971 - acc: 0.7046 - val_loss:
0.7320 - val_acc: 0.7049
Epoch 78/100
227468/227468 [==============================] - 5s 20us/step - loss: 0.6972 - acc: 0.7034 - val_loss:
0.7356 - val_acc: 0.6989
Epoch 79/100
227468/227468 [==============================] - 5s 20us/step - loss: 0.6972 - acc: 0.7053 - val_loss:
0.7326 - val_acc: 0.6985
Epoch 80/100
227468/227468 [==============================] - 5s 20us/step - loss: 0.6973 - acc: 0.7046 - val_loss:
0.7328 - val_acc: 0.6986
Epoch 81/100
227468/227468 [==============================] - 4s 19us/step - loss: 0.6969 - acc: 0.7049 - val_loss:
0.7314 - val_acc: 0.7094
Epoch 82/100
227468/227468 [==============================] - 4s 18us/step - loss: 0.6971 - acc: 0.7049 - val_loss:
0.7324 - val_acc: 0.7027
Epoch 83/100
227468/227468 [==============================] - 4s 19us/step - loss: 0.6970 - acc: 0.7053 - val_loss:
0.7349 - val_acc: 0.7080
Epoch 84/100
227468/227468 [==============================] - 4s 19us/step - loss: 0.6970 - acc: 0.7057 - val_loss:
0.7328 - val_acc: 0.7017
Epoch 85/100
227468/227468 [==============================] - 5s 20us/step - loss: 0.6968 - acc: 0.7047 - val_loss:
0.7319 - val_acc: 0.7055
Epoch 86/100
227468/227468 [==============================] - 4s 19us/step - loss: 0.6969 - acc: 0.7050 - val_loss:
0.7343 - val_acc: 0.7056
Epoch 87/100
227468/227468 [==============================] - 4s 18us/step - loss: 0.6968 - acc: 0.7061 - val_loss:
0.7315 - val_acc: 0.7090
Epoch 88/100
227468/227468 [==============================] - 5s 21us/step - loss: 0.6968 - acc: 0.7059 - val_loss:
0.7330 - val_acc: 0.7010
Epoch 89/100
227468/227468 [==============================] - 4s 19us/step - loss: 0.6967 - acc: 0.7052 - val_loss:
0.7318 - val_acc: 0.7028
Epoch 90/100
227468/227468 [==============================] - 5s 21us/step - loss: 0.6969 - acc: 0.7055 - val_loss:
0.7331 - val_acc: 0.7003
Epoch 91/100
227468/227468 [==============================] - 4s 19us/step - loss: 0.6969 - acc: 0.7058 - val_loss:
0.7326 - val_acc: 0.7050
Epoch 92/100
227468/227468 [==============================] - 4s 19us/step - loss: 0.6965 - acc: 0.7059 - val_loss:
0.7325 - val_acc: 0.7058
Epoch 93/100
227468/227468 [==============================] - 5s 20us/step - loss: 0.6968 - acc: 0.7059 - val_loss:
0.7314 - val_acc: 0.7091
Epoch 94/100
227468/227468 [==============================] - 5s 24us/step - loss: 0.6969 - acc: 0.7054 - val_loss:
0.7315 - val_acc: 0.7110
Epoch 95/100
227468/227468 [==============================] - 5s 21us/step - loss: 0.6968 - acc: 0.7060 - val_loss:
```