Time-series forecasting with deep learning & LSTM autoencoders

- The purpose of this work is to show one way time-series data can be effiently encoded to lower dimensions, to be used into non time-series models.
- Here I'll encode a time-series of size 12 (12 months) to a single value and use it on a MLP deep learning model, instead of using the time-series on a LSTM model that could be the regular approach.
- The first part of the data preparation is from my other kernel [Model stacking, feature engineering and EDA](#).
- This work was inspired by this Machinelearningmastery post [A Gentle Introduction to LSTM Autoencoders](#), make sure to check out.

Predict future sales

We are asking you to predict total sales for every product and store in the next month. By solving this competition you will be able to apply and enhance your data science skills.

You are provided with daily historical sales data. The task is to forecast the total amount of products sold in every shop for the test set. Note that the list of shops and products slightly changes every month. Creating a robust model that can handle such situations is part of the challenge.

Data fields description:

- ID - an Id that represents a (Shop, Item) tuple within the test set
- shop_id - unique identifier of a shop
- item_id - unique identifier of a product
- item_category_id - unique identifier of item category
- date_block_num - a consecutive month number, used for convenience. January 2013 is 0, February 2013 is 1,..., October 2015 is 33
- date - date in format dd/mm/yyyy
- item_cnt_day - number of products sold. You are predicting a monthly amount of this measure
- item_price - current price of an item
- item_name - name of item
- shop_name - name of shop
- item_category_name - name of item category

Dependencies

In [1]:

```
import os, warnings, random
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
import tensorflow as tf
import tensorflow.keras.layers as L
from tensorflow.keras import optimizers, Sequential, Model

# Set seeds to make the experiment more reproducible.
def seed_everything(seed=0):
    random.seed(seed)
    np.random.seed(seed)
    tf.random.set_seed(seed)
    os.environ['PYTHONHASHSEED'] = str(seed)
    os.environ['TF_DETERMINISTIC_OPS'] = '1'

seed = 0
seed_everything(seed)
```

```
warnings.filterwarnings('ignore')
pd.set_option('display.float_format', lambda x: '%.2f' % x)
```

Loading data

In [2]:

```
test = pd.read_csv('../input/test.csv', dtype={'ID': 'int32', 'shop_id': 'int32',
                                                'item_id': 'int32'})
item_categories = pd.read_csv('../input/item_categories.csv',
                              dtype={'item_category_name': 'str', 'item_category_id': 'i
nt32'})
items = pd.read_csv('../input/items.csv', dtype={'item_name': 'str', 'item_id': 'int32',

                                                 'item_category_id': 'int32'})
shops = pd.read_csv('../input/shops.csv', dtype={'shop_name': 'str', 'shop_id': 'int32'}
)
sales = pd.read_csv('../input/sales_train.csv', parse_dates=['date'],
                    dtype={'date': 'str', 'date_block_num': 'int32', 'shop_id': 'int32',

                      'item_id': 'int32', 'item_price': 'float32', 'item_cnt_day': 'int3
2'})
```

Join data sets

In [3]:

```
train = sales.join(items, on='item_id', rsuffix='_').join(shops, on='shop_id', rsuffix='
_').join(item_categories, on='item_category_id', rsuffix='_').drop(['item_id_', 'shop_id
_', 'item_category_id_'], axis=1)
```

Let's take a look at the raw data

In [4]:

```
print(f'Train rows: {train.shape[0]}')
print(f'Train columns: {train.shape[1]}')

display(train.head().T)
display(train.describe())
```

```
Train rows: 2935849
Train columns: 10
```

	0	1	2	3	4
date	2013-02-01 00:00:00	2013-03-01 00:00:00	2013-05-01 00:00:00	2013-06-01 00:00:00	2013-01-15 00:00:00
date_block_num	0	0	0	0	0
shop_id	59	25	25	25	25
item_id	22154	2552	2552	2554	2555
item_price	999.00	899.00	899.00	1709.05	1099.00
item_cnt_day	1	1	-1	1	1
item_name	ЯВЛЕНИЕ 2012 (BD)	DEEP PURPLE The House Of Blue Light LP	DEEP PURPLE The House Of Blue Light LP	DEEP PURPLE Who Do You Think We Are LP	DEEP PURPLE 30 Very Best Of 2CD (Фирм.)
item_category_id	37	58	58	58	56
shop_name	Ярославль ТЦ "Альтаир"	Москва ТРК "Атриум"	Москва ТРК "Атриум"	Москва ТРК "Атриум"	Москва ТРК "Атриум"
item_category_name	Кино - Blu-Ray	Музыка - Винил	Музыка - Винил	Музыка - Винил	Музыка - CD фирменного

производства

	0	1	2	3	4

	date_block_num	shop_id	item_id	item_price	item_cnt_day	item_category_id
count	2935849.00	2935849.00	2935849.00	2935849.00	2935849.00	2935849.00
mean	14.57	33.00	10197.23	890.62	1.24	40.00
std	9.42	16.23	6324.30	1726.44	2.62	17.10
min	0.00	0.00	0.00	-1.00	-22.00	0.00
25%	7.00	22.00	4476.00	249.00	1.00	28.00
50%	14.00	31.00	9343.00	399.00	1.00	40.00
75%	23.00	47.00	15684.00	999.00	1.00	55.00
max	33.00	59.00	22169.00	307980.00	2169.00	83.00

Time period of the dataset

In [5]:

```
print(f"Min date from train set: {train['date'].min().date()}")
print(f"Max date from train set: {train['date'].max().date()}")
```

```
Min date from train set: 2013-01-01
Max date from train set: 2015-12-10
```

I'm leaving only the "shop_id" and "item_id" that exist in the test set to have more accurate results.

In [6]:

```
test_shop_ids = test['shop_id'].unique()
test_item_ids = test['item_id'].unique()
# Only shops that exist in test set.
train = train[train['shop_id'].isin(test_shop_ids)]
# Only items that exist in test set.
train = train[train['item_id'].isin(test_item_ids)]
```

Data preprocessing

- **I'm dropping all features but "item_cnt_day" because I'll be using only it as a univariate time-series.**
- **We are asked to predict total sales for every product and store in the next month, and our data is given by day, so let's aggregate the data by month.**
- **Also I'm leaving only monthly "item_cnt" >= 0 and <= 20, as this seems to be the distributions of the test set.**

In [7]:

```
train_monthly = train[['date', 'date_block_num', 'shop_id', 'item_id', 'item_cnt_day']]
train_monthly = train_monthly.sort_values('date').groupby(['date_block_num', 'shop_id',
'item_id'], as_index=False)
train_monthly = train_monthly.agg({'item_cnt_day':['sum']})
train_monthly.columns = ['date_block_num', 'shop_id', 'item_id', 'item_cnt']
train_monthly = train_monthly.query('item_cnt >= 0 and item_cnt <= 20')
# Label
train_monthly['item_cnt_month'] = train_monthly.sort_values('date_block_num').groupby(['
shop_id', 'item_id'])['item_cnt'].shift(-1)

display(train_monthly.head(10).T)
display(train_monthly.describe().T)
```

	0	1	2	3	4	5	6	7	8	9
date_block_num	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
shop_id	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00

	0	1	2	3	4	5	6	7	8	9
item_id	33.00	482.00	491.00	839.00	1007.00	1010.00	1023.00	1204.00	1224.00	1247.00
item_cnt	1.00	1.00	1.00	1.00	3.00	1.00	2.00	1.00	1.00	1.00
item_cnt_month	2.00	1.00	1.00	1.00	1.00	1.00	1.00	nan	nan	nan

	count	mean	std	min	25%	50%	75%	max
date_block_num	593829.00	20.18	9.14	0.00	13.00	22.00	28.00	33.00
shop_id	593829.00	32.07	16.90	2.00	19.00	31.00	47.00	59.00
item_id	593829.00	10015.02	6181.82	30.00	4418.00	9171.00	15334.00	22167.00
item_cnt	593829.00	2.10	2.31	0.00	1.00	1.00	2.00	20.00
item_cnt_month	482536.00	2.07	2.17	0.00	1.00	1.00	2.00	20.00

Time-series processing

- **As I only need the "item_cnt" feature as a series, I can get that easily by just using a pivot operation.**
- **This way I'll also get the missing months from each "shop_id" and "item_id", and then replace them with 0 (otherwise would be "nan").**

In [8]:

```
monthly_series = train_monthly.pivot_table(index=['shop_id', 'item_id'], columns='date_bl
ock_num',values='item_cnt', fill_value=0).reset_index()
monthly_series.head()
```

Out[8]:

date_block_num	shop_id	item_id	0	1	2	3	4	5	6	7	...	24	25	26	27	28	29	30	31	32	33
0	2	30	0	0	1	0	0	1	0	0	...	0	0	0	0	0	0	0	0	0	0
1	2	31	0	4	1	1	0	0	0	0	...	0	0	0	0	0	0	0	0	0	1
2	2	32	0	0	0	0	0	0	0	0	...	0	0	1	0	0	0	0	1	0	0
3	2	33	1	0	0	0	0	0	0	0	...	0	1	0	1	1	0	1	0	1	0
4	2	53	0	0	0	0	0	0	0	0	...	0	1	0	0	0	0	0	0	0	0

5 rows × 36 columns

Currently I have one series (33 months) for each unique pair of "shop_id" and "item_id", but probably would be better to have multiple smaller series for each unique pair, so I'm generating multiple series of size 12 (one year) for each unique pair.

In [9]:

```
first_month = 20
last_month = 33
serie_size = 12
data_series = []

for index, row in monthly_series.iterrows():
    for month1 in range((last_month - (first_month + serie_size)) + 1):
        serie = [row['shop_id'], row['item_id']]
        for month2 in range(serie_size + 1):
            serie.append(row[month1 + first_month + month2])
        data_series.append(serie)

columns = ['shop_id', 'item_id']
[columns.append(i) for i in range(serie_size)]
columns.append('label')

data_series = pd.DataFrame(data_series, columns=columns)
data_series.head()
```

Out[9]:

	shop_id	item_id	0	1	2	3	4	5	6	7	8	9	10	11	label
0	2	30	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	30	0	0	0	0	0	0	0	0	0	0	0	0	0
2	2	31	0	0	0	0	0	0	0	0	0	0	0	0	0
3	2	31	0	0	0	0	0	0	0	0	0	0	0	0	1
4	2	32	2	2	0	2	0	0	1	0	0	0	0	1	0

Dropping identifier columns as we don't need them anymore.

In [10]:

```
data_series = data_series.drop(['item_id', 'shop_id'], axis=1)
```

Train and validation sets.

In [11]:

```
labels = data_series['label']
data_series.drop('label', axis=1, inplace=True)
train, valid, Y_train, Y_valid = train_test_split(data_series, labels.values, test_size=
0.10, random_state=0)
```

In [12]:

```
print("Train set", train.shape)
print("Validation set", valid.shape)
train.head()
```

```
Train set (200327, 12)
Validation set (22259, 12)
```

Out[12]:

	0	1	2	3	4	5	6	7	8	9	10	11
207604	0	0	0	0	0	0	0	0	0	0	0	0
45150	0	0	0	0	0	0	0	0	0	0	0	0
143433	0	0	4	2	1	2	2	1	0	0	0	1
202144	0	0	0	0	0	0	0	0	0	0	0	0
136088	0	0	0	0	0	0	0	1	0	0	1	0

Reshape data.

- Time-series shape **(data points, time-steps, features)**.

In [13]:

```
X_train = train.values.reshape((train.shape[0], train.shape[1], 1))
X_valid = valid.values.reshape((valid.shape[0], valid.shape[1], 1))

print("Train set reshaped", X_train.shape)
print("Validation set reshaped", X_valid.shape)
```

```
Train set reshaped (200327, 12, 1)
Validation set reshaped (22259, 12, 1)
```

First let's begin with how a regular RNN time-series approach could be.

Regular LSTM model.

In [14]:

```
serie_size =  X_train.shape[1] # 12
n_features =  X_train.shape[2] # 1

epochs = 20
batch = 128
lr = 0.0001

lstm_model = Sequential()
lstm_model.add(L.LSTM(10, input_shape=(serie_size, n_features), return_sequences=True))
lstm_model.add(L.LSTM(6, activation='relu', return_sequences=True))
lstm_model.add(L.LSTM(1, activation='relu'))
lstm_model.add(L.Dense(10, kernel_initializer='glorot_normal', activation='relu'))
lstm_model.add(L.Dense(10, kernel_initializer='glorot_normal', activation='relu'))
lstm_model.add(L.Dense(1))
lstm_model.summary()

adam = optimizers.Adam(lr)
lstm_model.compile(loss='mse', optimizer=adam)
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 12, 10)            480
_________________________________________________________________
lstm_1 (LSTM)                (None, 12, 6)             408
_________________________________________________________________
lstm_2 (LSTM)                (None, 1)                 32
_________________________________________________________________
dense (Dense)                (None, 10)                20
_________________________________________________________________
dense_1 (Dense)              (None, 10)                110
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 11
=================================================================
Total params: 1,061
Trainable params: 1,061
Non-trainable params: 0
_________________________________________________________________
```

In [15]:

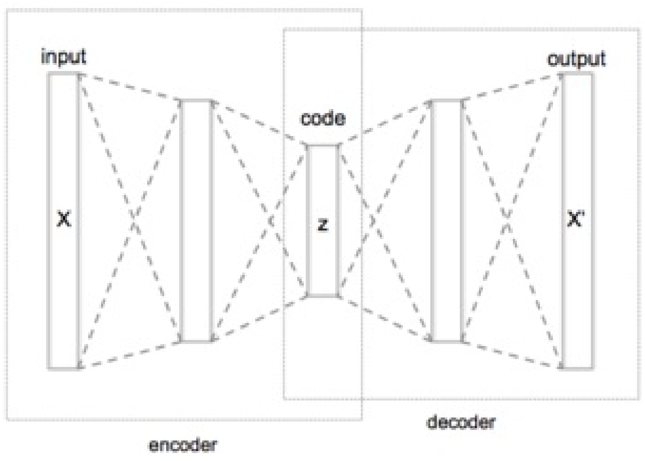
```
lstm_history = lstm_model.fit(X_train, Y_train,
                              validation_data=(X_valid, Y_valid),
                              batch_size=batch,
                              epochs=epochs,
                              verbose=2)
```

```
Epoch 1/20
1566/1566 - 74s - loss: 1.5067 - val_loss: 1.3122
Epoch 2/20
1566/1566 - 72s - loss: 1.3280 - val_loss: 1.2557
Epoch 3/20
1566/1566 - 75s - loss: 1.2744 - val_loss: 1.2234
Epoch 4/20
1566/1566 - 74s - loss: 1.2338 - val_loss: 1.1950
Epoch 5/20
1566/1566 - 72s - loss: 1.2027 - val_loss: 1.1726
Epoch 6/20
1566/1566 - 71s - loss: 1.1755 - val_loss: 1.1573
Epoch 7/20
1566/1566 - 75s - loss: 1.1700 - val_loss: 1.1529
Epoch 8/20
1566/1566 - 74s - loss: 1.1676 - val_loss: 1.1511
Epoch 9/20
1566/1566 - 72s - loss: 1.1664 - val_loss: 1.1484
Epoch 10/20
```

```
1566/1566 - 72s - loss: 1.1652 - val_loss: 1.1484
Epoch 11/20
1566/1566 - 70s - loss: 1.1642 - val_loss: 1.1489
Epoch 12/20
1566/1566 - 71s - loss: 1.1637 - val_loss: 1.1464
Epoch 13/20
1566/1566 - 73s - loss: 1.1629 - val_loss: 1.1453
Epoch 14/20
1566/1566 - 72s - loss: 1.1624 - val_loss: 1.1446
Epoch 15/20
1566/1566 - 72s - loss: 1.1616 - val_loss: 1.1444
Epoch 16/20
1566/1566 - 72s - loss: 1.1616 - val_loss: 1.1459
Epoch 17/20
1566/1566 - 71s - loss: 1.1610 - val_loss: 1.1430
Epoch 18/20
1566/1566 - 74s - loss: 1.1604 - val_loss: 1.1421
Epoch 19/20
1566/1566 - 77s - loss: 1.1604 - val_loss: 1.1458
Epoch 20/20
1566/1566 - 72s - loss: 1.1603 - val_loss: 1.1410
```

Autoencoder

- **Now we will build an autoencoder to learn how to reconstruct the input, this way it internally learns the best way to represent the input in lower dimensions.**
- **The reconstruct model is composed of an encoder and a decoder, the encoder is responsible for learning how to represent the input into lower dimensions and the decoder learns how to rebuild the smaller representations into the input again.**
- **Here is a structural representations of an autoencoder:**



- **After the models is trained we can keep only the encoder part and we'll have a model that is able to do what we want.**

LSTM Autoencoder.

In [16]:

```
encoder_decoder = Sequential()
encoder_decoder.add(L.LSTM(serie_size, activation='relu', input_shape=(serie_size, n_fea
tures), return_sequences=True))
encoder_decoder.add(L.LSTM(6, activation='relu', return_sequences=True))
encoder_decoder.add(L.LSTM(1, activation='relu'))
encoder_decoder.add(L.RepeatVector(serie_size))
encoder_decoder.add(L.LSTM(serie_size, activation='relu', return_sequences=True))
encoder_decoder.add(L.LSTM(6, activation='relu', return_sequences=True))
encoder_decoder.add(L.TimeDistributed(L.Dense(1)))
encoder_decoder.summary()

adam = optimizers.Adam(lr)
```

```
encoder_decoder.compile(loss='mse', optimizer=adam)
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
lstm_3 (LSTM)                (None, 12, 12)            672       
_________________________________________________________________
lstm_4 (LSTM)                (None, 12, 6)             456       
_________________________________________________________________
lstm_5 (LSTM)                (None, 1)                 32        
_________________________________________________________________
repeat_vector (RepeatVector) (None, 12, 1)             0         
_________________________________________________________________
lstm_6 (LSTM)                (None, 12, 12)            672       
_________________________________________________________________
lstm_7 (LSTM)                (None, 12, 6)             456       
_________________________________________________________________
time_distributed (TimeDistri (None, 12, 1)             7         
=================================================================
Total params: 2,295
Trainable params: 2,295
Non-trainable params: 0
_________________________________________________________________
```

In [17]:

```
encoder_decoder_history = encoder_decoder.fit(X_train, X_train,
                                              batch_size=batch,
                                              epochs=epochs,
                                              verbose=2)
```

```
Epoch 1/20
1566/1566 - 166s - loss: 1.5515
Epoch 2/20
1566/1566 - 165s - loss: 1.1077
Epoch 3/20
1566/1566 - 167s - loss: 1.0431
Epoch 4/20
1566/1566 - 162s - loss: 1.0177
Epoch 5/20
1566/1566 - 159s - loss: 1.0008
Epoch 6/20
1566/1566 - 161s - loss: 0.9840
Epoch 7/20
1566/1566 - 162s - loss: 0.9694
Epoch 8/20
1566/1566 - 169s - loss: 0.9595
Epoch 9/20
1566/1566 - 174s - loss: 0.9518
Epoch 10/20
1566/1566 - 171s - loss: 0.9471
Epoch 11/20
1566/1566 - 168s - loss: 0.9446
Epoch 12/20
1566/1566 - 168s - loss: 0.9388
Epoch 13/20
1566/1566 - 165s - loss: 0.9340
Epoch 14/20
1566/1566 - 164s - loss: 0.9323
Epoch 15/20
1566/1566 - 161s - loss: 0.9298
Epoch 16/20
1566/1566 - 160s - loss: 0.9282
Epoch 17/20
1566/1566 - 163s - loss: 0.9232
Epoch 18/20
1566/1566 - 163s - loss: 0.9172
Epoch 19/20
1566/1566 - 161s - loss: 0.9178
Epoch 20/20
1566/1566 - 163s - loss: 0.9164
```

You should be aware that the better the autoencoder is able to reconstruct the input the better it internally encodes the input, in other words if we have a good autoencoder we probably will have an equally good encoder.

Let's take a look at the layers of the encoder_decoder model:

In [18]:

```
rpt_vector_layer = Model(inputs=encoder_decoder.inputs, outputs=encoder_decoder.layers[3
].output)
time_dist_layer = Model(inputs=encoder_decoder.inputs, outputs=encoder_decoder.layers[5]
.output)
encoder_decoder.layers
```

Out[18]:

```
[<tensorflow.python.keras.layers.recurrent_v2.LSTM at 0x7f89eec7d710>,
 <tensorflow.python.keras.layers.recurrent_v2.LSTM at 0x7f89f7387ad0>,
 <tensorflow.python.keras.layers.recurrent_v2.LSTM at 0x7f89ec1c1590>,
 <tensorflow.python.keras.layers.core.RepeatVector at 0x7f89e7e34250>,
 <tensorflow.python.keras.layers.recurrent_v2.LSTM at 0x7f89ec15c190>,
 <tensorflow.python.keras.layers.recurrent_v2.LSTM at 0x7f89ec0c4c10>,
 <tensorflow.python.keras.layers.wrappers.TimeDistributed at 0x7f89ec0df990>]
```

About the autoencoder layers

LSTM

- **This is just a regular LSTM layer, a layer that is able to receive sequence data and learn based on it nothing much to talk about.**

RepeatVector layer

- **Here is something we don't usually see, this layers basically repeats it's input "n" times, the reason to use it is because the last layers from the encoder part (the layer with one neuron) don't return sequences, so it does not outputs a sequenced data, this way we can't just add another LSTM layer after it, we need a way to turn this output into a sequence of the same time-steps of the model input, this is where "RepeatVector" layers comes in.**
- **Let's see what it outputs.**

In [19]:

```
rpt_vector_layer_output = rpt_vector_layer.predict(X_train[:1])
print('Repeat vector output shape', rpt_vector_layer_output.shape)
print('Repeat vector output sample')
print(rpt_vector_layer_output[0])
```

```
Repeat vector output shape (1, 12, 1)
Repeat vector output sample
[[1.7652086]
 [1.7652086]
 [1.7652086]
 [1.7652086]
 [1.7652086]
 [1.7652086]
 [1.7652086]
 [1.7652086]
 [1.7652086]
 [1.7652086]
 [1.7652086]
 [1.7652086]]
```

As you can see this is just the same value repeated some times to match the same shape of the model input.

TimeDistributed layer

- **This layer is more common, sometimes is used when you want to mix RNN layers with other kind of layers.**
- **We could output the model with another LSTM layer with one neuron and "return_sequences=True" parameter, but using a "TimeDistributed" layer wrapping a "Dense" layer we will have the same weights for each outputted time-step.**

In [20]:

```
time_dist_layer_output = time_dist_layer.predict(X_train[:1])
print('Time distributed output shape', time_dist_layer_output.shape)
print('Time distributed output sample')
print(time_dist_layer_output[0])
```

```
Time distributed output shape (1, 12, 6)
Time distributed output sample
[[1.9910030e-02 0.0000000e+00 3.8524549e-03 7.4686572e-02 1.1851047e-01
  6.6910662e-02]
 [1.0625558e-02 0.0000000e+00 3.5476349e-02 1.1288490e-01 2.1432281e-01
  1.5205307e-01]
 [4.8153275e-03 0.0000000e+00 8.4371455e-02 7.0393331e-02 2.9059261e-01
  2.0324098e-01]
 [1.9158795e-03 0.0000000e+00 1.3351467e-01 3.9786749e-02 3.4163573e-01
  2.0546621e-01]
 [7.1840204e-04 0.0000000e+00 1.7135502e-01 2.1733187e-02 3.6997330e-01
  1.9290383e-01]
 [2.6656259e-04 0.0000000e+00 1.9970903e-01 1.1735361e-02 3.8098180e-01
  1.9020893e-01]
 [9.8010998e-05 2.8399068e-03 2.2207233e-01 6.2495852e-03 3.8081640e-01
  1.9687831e-01]
 [3.5544392e-05 9.2054876e-03 2.4055974e-01 3.2736955e-03 3.7324408e-01
  2.0759062e-01]
 [1.2677572e-05 1.7838215e-02 2.5607160e-01 1.6867360e-03 3.6060077e-01
  2.1849820e-01]
 [4.4451099e-06 2.7221033e-02 2.6882043e-01 8.5621356e-04 3.4438208e-01
  2.2764295e-01]
 [1.5344931e-06 3.6003374e-02 2.7878711e-01 4.2922908e-04 3.2555303e-01
  2.3437236e-01]
 [5.2304472e-07 4.3212134e-02 2.8595468e-01 2.1306782e-04 3.0475688e-01
  2.3878156e-01]]
```

Another good explanation about the used layers

Defining the encoding model.

- **What I want is to encode the whole series into a single value, so I need the output from the layer with a single neuron (in this case it's the third LSTM layer).**
- **I'll take only the encoding part of the model and define it as a new one.**

In [21]:

```
encoder = Model(inputs=encoder_decoder.inputs, outputs=encoder_decoder.layers[2].output)
```

Now let's encode the train and validation time-series.

In [22]:

```
train_encoded = encoder.predict(X_train)
validation_encoded = encoder.predict(X_valid)
print('Encoded time-series shape', train_encoded.shape)
print('Encoded time-series sample', train_encoded[0])
```

```
Encoded time-series shape (200327, 1)
Encoded time-series sample [1.7652105]
```

Add new encoded features to the train and validation sets.

In [23]:

```
train['encoded'] = train_encoded
train['label'] = Y_train

valid['encoded'] = validation_encoded
valid['label'] = Y_valid

train.head(10)
```

Out[23]:

	0	1	2	3	4	5	6	7	8	9	10	11	encoded	label
207604	0	0	0	0	0	0	0	0	0	0	0	0	1.77	0
45150	0	0	0	0	0	0	0	0	0	0	0	0	1.77	0
143433	0	0	4	2	1	2	2	1	0	0	0	1	60.37	1
202144	0	0	0	0	0	0	0	0	0	0	0	0	1.77	0
136088	0	0	0	0	0	0	0	1	0	0	1	0	2.93	1
121675	0	0	0	0	0	1	0	0	0	0	0	0	2.35	0
185281	0	0	0	0	0	0	0	0	0	0	0	0	1.77	1
70087	0	0	0	0	0	0	0	0	3	0	1	3	7.53	0
105249	0	0	0	0	0	0	0	0	0	0	0	0	1.77	0
183257	0	0	0	0	0	3	0	1	0	0	0	1	4.39	0

Now we can use the new encoded feature that is a representation of the whole time-series and train a "less complex" model that does not receives sequenced data as input.

MLP with LSTM encoded feature

- **For the MLP model I'm only using the current month "item_count" and the encoded time-series feature from our LSTM encoder model, the idea is that we won't need the whole series because we already have a column that represents the whole series into a single value (it's like a dimensionality reduction).**

In [24]:

```
last_month = serie_size - 1
Y_train_encoded = train['label']
train.drop('label', axis=1, inplace=True)
X_train_encoded = train[[last_month, 'encoded']]

Y_valid_encoded = valid['label']
valid.drop('label', axis=1, inplace=True)
X_valid_encoded = valid[[last_month, 'encoded']]

print("Train set", X_train_encoded.shape)
print("Validation set", X_valid_encoded.shape)
```

```
Train set (200327, 2)
Validation set (22259, 2)
```

In [25]:

```
X_train_encoded.head()
```

Out[25]:

	11	encoded
207604	0	1.77
45150	0	1.77

	11	encoded
143433	1	60.37
202144	0	1.77
136088	0	2.93

In [26]:

```
mlp_model = Sequential()
mlp_model.add(L.Dense(10, kernel_initializer='glorot_normal', activation='relu', input_di
m=X_train_encoded.shape[1]))
mlp_model.add(L.Dense(10, kernel_initializer='glorot_normal', activation='relu'))
mlp_model.add(L.Dense(1))
mlp_model.summary()

adam = optimizers.Adam(lr)
mlp_model.compile(loss='mse', optimizer=adam)
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_4 (Dense)              (None, 10)                30        
_________________________________________________________________
dense_5 (Dense)              (None, 10)                110       
_________________________________________________________________
dense_6 (Dense)              (None, 1)                 11        
=================================================================
Total params: 151
Trainable params: 151
Non-trainable params: 0
_________________________________________________________________
```

In [27]:

```
mlp_history = mlp_model.fit(X_train_encoded.values, Y_train_encoded.values, epochs=epoch
s, batch_size=batch, validation_data=(X_valid_encoded, Y_valid_encoded), verbose=2)
```

```
Epoch 1/20
1566/1566 - 3s - loss: 27.5866 - val_loss: 1.8983
Epoch 2/20
1566/1566 - 3s - loss: 1.5284 - val_loss: 1.2943
Epoch 3/20
1566/1566 - 3s - loss: 1.2642 - val_loss: 1.2014
Epoch 4/20
1566/1566 - 3s - loss: 1.2394 - val_loss: 1.2023
Epoch 5/20
1566/1566 - 3s - loss: 1.2396 - val_loss: 1.2089
Epoch 6/20
1566/1566 - 3s - loss: 1.2395 - val_loss: 1.1996
Epoch 7/20
1566/1566 - 3s - loss: 1.2391 - val_loss: 1.1996
Epoch 8/20
1566/1566 - 3s - loss: 1.2363 - val_loss: 1.2074
Epoch 9/20
1566/1566 - 3s - loss: 1.2322 - val_loss: 1.1920
Epoch 10/20
1566/1566 - 3s - loss: 1.2287 - val_loss: 1.1903
Epoch 11/20
1566/1566 - 3s - loss: 1.2265 - val_loss: 1.1878
Epoch 12/20
1566/1566 - 3s - loss: 1.2261 - val_loss: 1.1872
Epoch 13/20
1566/1566 - 4s - loss: 1.2243 - val_loss: 1.1873
Epoch 14/20
1566/1566 - 3s - loss: 1.2238 - val_loss: 1.1961
Epoch 15/20
1566/1566 - 3s - loss: 1.2223 - val_loss: 1.1892
Epoch 16/20
1566/1566 - 3s - loss: 1.2209 - val_loss: 1.1841
Epoch 17/20
1566/1566 - 3s - loss: 1.2195 - val_loss: 1.1839
```

```
Epoch 18/20
1566/1566 - 3s - loss: 1.2186 - val_loss: 1.1954
Epoch 19/20
1566/1566 - 3s - loss: 1.2176 - val_loss: 1.1901
Epoch 20/20
1566/1566 - 3s - loss: 1.2166 - val_loss: 1.1797
```

Comparing models

- **As you can see I tried to build both models with a similar topology (type/number of layers and neurons), so it could make more sense to compare them.**
- **The results are pretty close, also they may change a bit depending on the random initialization of the networks weights, so I would say they are very similar in terms of performance.**

Model training

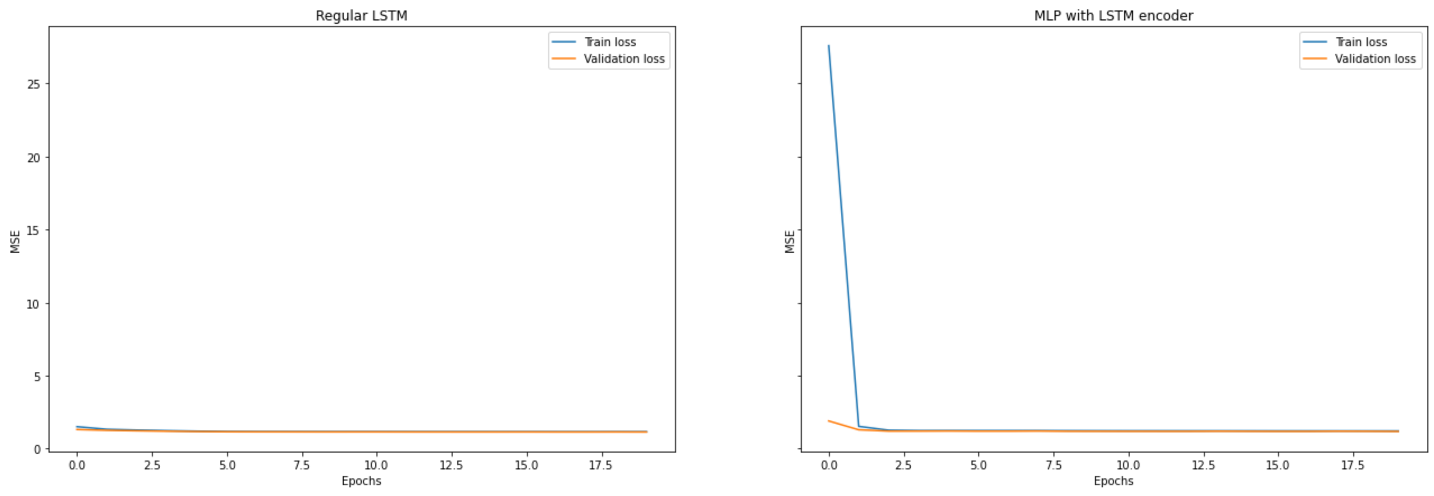
In [28]:

```
fig, (ax1, ax2) = plt.subplots(1, 2, sharey=True,figsize=(22,7))

ax1.plot(lstm_history.history['loss'], label='Train loss')
ax1.plot(lstm_history.history['val_loss'], label='Validation loss')
ax1.legend(loc='best')
ax1.set_title('Regular LSTM')
ax1.set_xlabel('Epochs')
ax1.set_ylabel('MSE')

ax2.plot(mlp_history.history['loss'], label='Train loss')
ax2.plot(mlp_history.history['val_loss'], label='Validation loss')
ax2.legend(loc='best')
ax2.set_title('MLP with LSTM encoder')
ax2.set_xlabel('Epochs')
ax2.set_ylabel('MSE')

plt.show()
```



Regular LSTM on train and validation.

In [29]:

```
lstm_train_pred = lstm_model.predict(X_train)
lstm_val_pred = lstm_model.predict(X_valid)
print('Train rmse:', np.sqrt(mean_squared_error(Y_train, lstm_train_pred)))
print('Validation rmse:', np.sqrt(mean_squared_error(Y_valid, lstm_val_pred)))
```

```
Train rmse: 1.076470938136834
Validation rmse: 1.0681919213608853
```

MLP with LSTM encoder on train and validation.

In [30]:

```
mlp_train_pred2 = mlp_model.predict(X_train_encoded.values)
mlp_val_pred2 = mlp_model.predict(X_valid_encoded.values)
print('Train rmse:', np.sqrt(mean_squared_error(Y_train_encoded, mlp_train_pred2)))
print('Validation rmse:', np.sqrt(mean_squared_error(Y_valid_encoded, mlp_val_pred2)))
```

```
Train rmse: 1.1017709424914062
Validation rmse: 1.0861355726433715
```

Build test set

Since we have two models I'll build test sets to apply on both of them.

In [31]:

```
latest_records = monthly_series.drop_duplicates(subset=['shop_id', 'item_id'])
X_test = pd.merge(test, latest_records, on=['shop_id', 'item_id'], how='left', suffixes=
['', '_'])
X_test.fillna(0, inplace=True)
X_test.drop(['ID', 'item_id', 'shop_id'], axis=1, inplace=True)
X_test.head()
```

Out[31]:

	0	1	2	3	4	5	6	7	8	9	...	24	25	26	27	28	29	30	31	32	33
0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	...	2.00	0.00	0.00	0.00	1.00	1.00	1.00	3.00	1.00	0.00
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	...	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	...	0.00	0.00	0.00	0.00	3.00	2.00	0.00	1.00	3.00	1.00
3	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	...	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	...	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

5 rows × 34 columns

Regular LSTM model test predictions

- For the regular LSTM model we just need the last 12 months, because that's our series input size.

In [32]:

```
X_test = X_test[[(i + (34 - serie_size)) for i in range(serie_size)]]
X_test.head()
```

Out[32]:

	22	23	24	25	26	27	28	29	30	31	32	33
0	1.00	2.00	2.00	0.00	0.00	0.00	1.00	1.00	1.00	3.00	1.00	0.00
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2	0.00	0.00	0.00	0.00	0.00	0.00	3.00	2.00	0.00	1.00	3.00	1.00
3	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Reshape data.

- Time-series shape **(data points, time-steps, features)**.

In [33]:

```
X_test_reshaped = X_test.values.reshape((X_test.shape[0], X_test.shape[1], 1))
print(X_test_reshaped.shape)
```

```
(214200, 12, 1)
```

Making predictions.

In [34]:

```
lstm_test_pred = lstm_model.predict(X_test_reshaped)
```

MLP with LSTM encoded feature test predictions

- **For the MLP model with the encoded features I'm only using the current month "item_count" and the encoded time-series feature from our LSTM encoder model.**

Encoding the time-series

In [35]:

```
test_encoded = encoder.predict(X_test_reshaped)
```

Add encoded features to the test set.

In [36]:

```
X_test['encoded'] = test_encoded
X_test.head()
```

Out[36]:

	22	23	24	25	26	27	28	29	30	31	32	33	encoded
0	1.00	2.00	2.00	0.00	0.00	0.00	1.00	1.00	1.00	3.00	1.00	0.00	4.88
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.77
2	0.00	0.00	0.00	0.00	0.00	0.00	3.00	2.00	0.00	1.00	3.00	1.00	6.93
3	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	2.27
4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.77

In [37]:

```
X_test_encoded = X_test[[33, 'encoded']]
print("Train set", X_test_encoded.shape)
X_test_encoded.head()
```

```
Train set (214200, 2)
```

Out[37]:

	33	encoded
0	0.00	4.88
1	0.00	1.77
2	1.00	6.93
3	0.00	2.27
4	0.00	1.77

Making predictions.

In [38]:

```
mlp_test_pred = mlp_model.predict(X_test_encoded)
```

Predictions from the regular LSTM model.

In [39]:

```
lstm_prediction = pd.DataFrame(test['ID'], columns=['ID'])
lstm_prediction['item_cnt_month'] = lstm_test_pred.clip(0., 20.)
lstm_prediction.to_csv('lstm_submission.csv', index=False)
lstm_prediction.head(10)
```

Out[39]:

	ID	item_cnt_month
0	0	0.55
1	1	0.49
2	2	0.85
3	3	0.17
4	4	0.49
5	5	0.44
6	6	0.98
7	7	0.17
8	8	1.22
9	9	0.49

Predictions from the MLP model with LSTM encodded feature .

In [40]:

```
mlp_prediction = pd.DataFrame(test['ID'], columns=['ID'])
mlp_prediction['item_cnt_month'] = mlp_test_pred.clip(0., 20.)
mlp_prediction.to_csv('mlp_submission.csv', index=False)
mlp_prediction.head(10)
```

Out[40]:

	ID	item_cnt_month
0	0	0.24
1	1	0.36
2	2	0.41
3	3	0.28
4	4	0.36
5	5	0.40
6	6	0.87
7	7	0.24
8	8	0.73
9	9	0.36

Just a disclaimer, you absolutely can get better results on any of the used models, I did not spent too much time tuning the models hyper parameters, as this is just for demonstration purpose, so if you want to give the code a try, you should surely tune a little more, if you get better results or any good insight about the models or architecture please let me know.

If you want to check out some interesting different approaches on time series problems take a look at this kernel Deep Learning for Time Series Forecasting

kernel Deep Learning for Time Series Forecasting .