Neural Network Analysis

In this project, was built an initial neural network (shallow) used to predict pass through medicines. Was provided a relation of 2 quarter (7 months) of stock with ARIMA prediction & a quarter prediction (2019/03).

In [1]:

```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import copy
import progressbar
import numpy as np
import pandas as pd
from datetime import datetime
import matplotlib.pyplot as plt
from matplotlib.pyplot import *
```

In [2]:

```
# Production only
# from google.colab import drive
# drive.mount('/content/drive')
```

Load and prepare the data

A critical step in working with neural networks is preparing the data correctly. Variables on different scales make it difficult for the network to efficiently learn the correct weights. Below, we've written the code to load and prepare the data. You'll learn more about this soon!

In [3]:

```
arimaset_legend = '(ARIMA)QUETIAPINA'
microset_legend = '(MICROSET)QUETIAPINA'

# Production Path
# arimaset_path = '/content/drive/My Drive/DAF-USP/Preditivo/Colab Notebooks/dat
a/arimaset-quetiapina-alagoas.csv'
# microset_path = '/content/drive/My Drive/DAF-USP/Preditivo/Colab Notebooks/dat
a/microset-quetiapina-alagoas.csv'

arimaset_path = 'datasets/arimaset-quetiapina-alagoas.csv'
microset_path = 'datasets/microset-quetiapina-alagoas.csv'

arimaset_df = pd.read_csv(arimaset_path)
microset_df = pd.read_csv(microset_path)

arimaset_df['DT'] = pd.to_datetime(arimaset_df['DT'], format='%Y%m', errors='ign
ore')
microset_df['DT'] = pd.to_datetime(microset_df['DT'], format='%Y%m', errors='ign
ore')
```

In [4]:

```
arimaset_df.head()
```

Out[4]:

	DT	MES	QTD
0	2013-01-01	1	63932
1	2013-01-01	1	116474
2	2013-01-01	1	71167
3	2013-01-01	1	1306
4	2013-01-01	1	62842

In [5]:

```
microset_df.head()
```

Out[5]:

	DT	MES	QTD
0	2011-12-01	12	390
1	2012-01-01	1	1500
2	2012-02-01	2	1875
3	2012-03-01	3	2785
4	2012-04-01	4	3525

In [6]:

```
print('Ajusted QTD (DAF has 3% of Data)')
microset_df.QTD = microset_df.QTD * 33.33
microset_df.head()
```

```
Ajusted QTD (DAF has 3% of Data)
```

Out[6]:

	DT	MES	QTD
0	2011-12-01	12	12998.70
1	2012-01-01	1	49995.00
2	2012-02-01	2	62493.75
3	2012-03-01	3	92824.05
4	2012-04-01	4	117488.25

Checking out the data

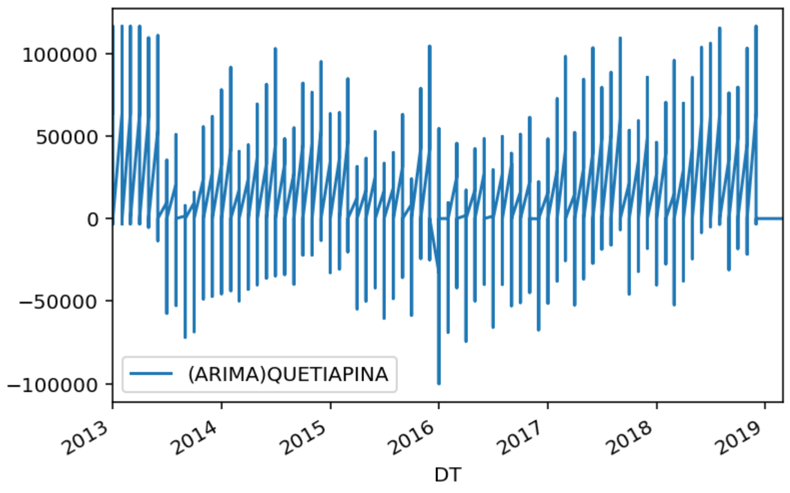
This dataset has the number of medicines acquired by Health Ministery for each month of each day from January 1 2017 to November 31 2018. The number of medicines is split between casual and registered, summed up in the `count1` column. You can see the first few rows of the data above.

In [7]:

```
fig, ax = subplots()
arimaset_df[:].plot(x='DT', y=['QTD'], ax=ax)
ax.legend([arimaset_legend])
```

Out[7]:

```
<matplotlib.legend.Legend at 0x10fd76290>
```



DBSet with Outlier

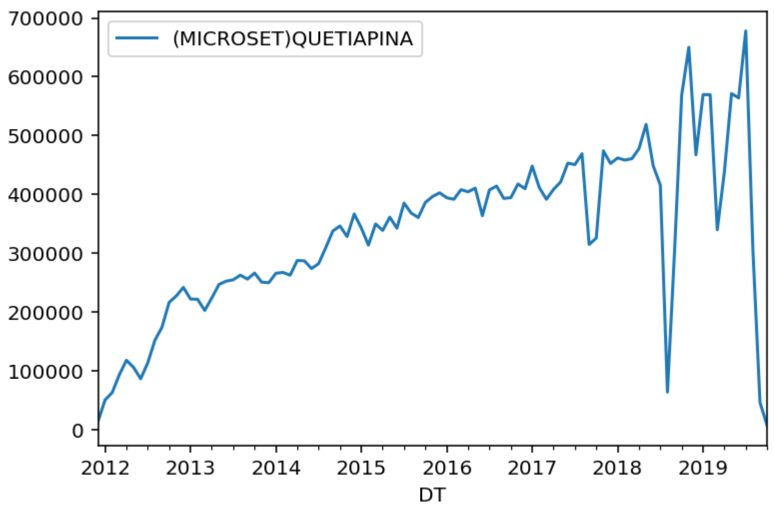
In [8]:

```
fig, ax = subplots()
# microset_df = microset_df[microset_df.DT < datetime(2017,9,1)]
microset_df[:].plot(x='DT', y=['QTD'], ax=ax)

ax.legend([microset_legend])
```

Out[8]:

```
<matplotlib.legend.Legend at 0x1144aa890>
```



Outlier Clear

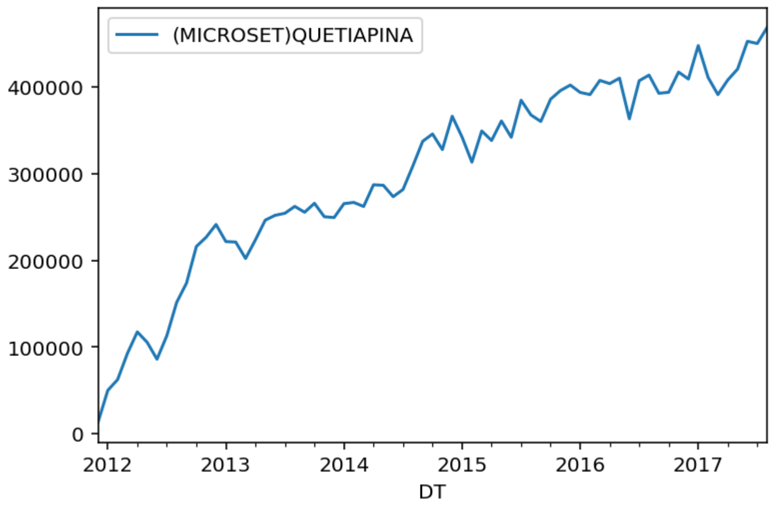
In [9]:

```
fig, ax = subplots()
microset_df = microset_df[microset_df.DT < datetime(2017,9,1)]
microset_df[:].plot(x='DT', y=['QTD'], ax=ax)

ax.legend([microset_legend])
```

Out[9]:

```
<matplotlib.legend.Legend at 0x10fd67e10>
```



Dummy variables

Here we have some categorical variables like **predicted** and **delivered**. To include these variables is necessary to make it binary dummy variables. This is simple to do with Pandas thanks to `get_dummies()` .

In [10]:

```
# Convert dtmonth to index month ignoring it year
# medicines['dtmonth'] = medicines['dtmonth'].apply(lambda x: int(x.split('-')
[1]))

dummy_fields = ['DT']
for each in dummy_fields:
    dummies = pd.get_dummies(microset_df[each], prefix=each, drop_first=False)
    microset = pd.concat([microset_df, dummies], axis=1)


fields_to_drop = ['DT']
data = microset.drop(fields_to_drop, axis=1)
data.head()
```

Out[10]:

	MES	QTD	DT_2011-12-01 00:00:00	DT_2012-01-01 00:00:00	DT_2012-02-01 00:00:00	DT_2012-03-01 00:00:00	DT_2012-04-01 00:00:00	DT_2012-05-01 00:00:00	DT_2012-06-01 00:00:00
0	12	12998.70	1	0	0	0	0	0	0
1	1	49995.00	0	1	0	0	0	0	0
2	2	62493.75	0	0	1	0	0	0	0
3	3	92824.05	0	0	0	1	0	0	0
4	4	117488.25	0	0	0	0	1	0	0

5 rows × 71 columns

In [11]:

```
dummy_fields = ['DT']
for each in dummy_fields:
    dummies = pd.get_dummies(arimaset_df[each], prefix=each, drop_first=False)
    arimaset = pd.concat([arimaset_df, dummies], axis=1)


fields_to_drop = ['DT']
arima_data = arimaset.drop(fields_to_drop, axis=1)
arima_data.head()
```

Out[11]:

	MES	QTD	DT_2013-01-01 00:00:00	DT_2013-02-01 00:00:00	DT_2013-03-01 00:00:00	DT_2013-04-01 00:00:00	DT_2013-05-01 00:00:00	DT_2013-06-01 00:00:00	DT_2013-07-01 00:00:00	D
0	1	63932	1	0	0	0	0	0	0	
1	1	116474	1	0	0	0	0	0	0	
2	1	71167	1	0	0	0	0	0	0	
3	1	1306	1	0	0	0	0	0	0	
4	1	62842	1	0	0	0	0	0	0	

5 rows × 77 columns

Scaling target variables

To make training the network easier, we'll standardize each of the continuous variables. That is, we'll shift and scale the variables such that they have zero mean and a standard deviation of 1.

The scaling factors are saved so we can go backwards when we use the network for predictions.

In [12]:

```
quant_features = ['MES', 'QTD']
# Store scalings in a dictionary so we can convert back later
scaled_features = {}
for each in quant_features:
    mean, std = data[each].mean(), data[each].std()
    scaled_features[each] = [mean, std]
    data.loc[:, each] = (data[each] - mean)/std

print('passed')
```

```
passed
```

In [13]:

```
quant_features = ['MES', 'QTD']
# Store scalings in a dictionary so we can convert back later
scaled_features = {}
for each in quant_features:
    mean, std = arima_data[each].mean(), arima_data[each].std()
    scaled_features[each] = [mean, std]
    arima_data.loc[:, each] = (arima_data[each] - mean)/std

print('passed')
```

```
passed
```

Splitting the data into training, testing, and validation sets

We'll save the data for the last approximately 2 months to use as a test set after we've trained the network. We'll use this set to make predictions and compare them with the actual number of medicines.

In [14]:

```
# goal_data = daf_df['QTD']
# features_data = daf_df.loc[:, daf_df.columns != 'QTD']


# trainset, testset, validateset = np.split(daf_df.sample(frac=1), [int(.7*len(d
af_df)), int(.8*len(daf_df))])
# X_trainset, X_testset, X_validateset = np.split(features_data.sample(frac=1),
 [int(.7*len(features_data)), int(.8*len(features_data))])
# Y_trainset, Y_testset, Y_validateset = np.split(goal_data.sample(frac=1), [int
(.7*len(goal_data)), int(.8*len(goal_data))])



# Save data for approximately the last 2 months
test_data = data[:] # test_data = data[-1*8:]

# Now remove the test data from the data set
data = data[:] #data = data[:-1*8]

# Separate the data into features and targets
target_fields = ['QTD']
features, targets = data.drop(target_fields, axis=1), data[target_fields] # feat
ures(inputs) = data except targets
test_features, test_targets = test_data.drop(target_fields, axis=1), test_data[t
arget_fields]
# arima_targets = arimaset_df[(arimaset_df.DT > datetime(2017,1,1)) & (arimaset_
df.DT < datetime(2017,9,1))][target_fields]
# arima_targets = arima_targets[-1*8:]
arima_targets = arima_data[-1*8:]

test_data.head()
```

Out[14]:

	MES	QTD	DT_2011-12-01 00:00:00	DT_2012-01-01 00:00:00	DT_2012-02-01 00:00:00	DT_2012-03-01 00:00:00	DT_2012-04-01 00:00:00	DT_2012-05-01 00:00:00	DT_2 0 00:0
0	1.629993	-2.608054	1	0	0	0	0	0	
1	-1.542224	-2.273090	0	1	0	0	0	0	
2	-1.253841	-2.159926	0	0	1	0	0	0	
3	-0.965457	-1.885316	0	0	0	1	0	0	
4	-0.677074	-1.662006	0	0	0	0	1	0	

5 rows × 71 columns

In [15]:

```
arima_data.head()
```

Out[15]:

	MES	QTD	DT_2013-01-01 00:00:00	DT_2013-02-01 00:00:00	DT_2013-03-01 00:00:00	DT_2013-04-01 00:00:00	DT_2013-05-01 00:00:00	DT_2013-06-01 00:00:00	DT_2 0 00:0
0	-1.547998	1.146847	1	0	0	0	0	0	
1	-1.547998	2.605421	1	0	0	0	0	0	
2	-1.547998	1.347691	1	0	0	0	0	0	
3	-1.547998	-0.591661	1	0	0	0	0	0	
4	-1.547998	1.116588	1	0	0	0	0	0	

5 rows × 77 columns

We'll split the data into two sets, one for training and one for validating as the network is being trained. Since this is time series data, we'll train on historical data, then try to predict on future data (the validation set).

In [16]:

```
# Hold out the last 8 months or so of the remaining data as a validation set
train_features, train_targets = features[:-2*8], targets[:-2*8]
val_features, val_targets = features[-2*8:], targets[-2*8:]
```

Time to build the network

Below you'll build your network. We've built out the structure and the backwards pass. **You'll implement** the **forward pass** through the network. **You'll** also **set** the hyperparameters: **the learning rate**, the **number of hidden units**, and the **number of training passes**.

The network has two layers: a **hidden** layer and an **output** layer. The **hidden** layer will use the **sigmoid function for activations**. The **output** layer has only one node and is **used for the regression**, the **output** of the node **is the same as the input** of the node. That is, **the activation function is** $f(x) = x$. A function that takes the input signal and generates an output signal, but takes into account the threshold, is called an activation function. We work through each layer of our network calculating the outputs for each neuron. All of the **outputs from one layer become inputs to the neurons on the next layer**. This process is **called *forward propagation***.

We **use** the **weights** to **propagate signals forward** from the **input to the output** layers in a neural network. We use the **weights** to **also propagate error backwards** from the **output back** into the network to update our weights. This is called ***backpropagation***.

> **Hint:** You'll need the derivative of the output activation function ($f(x) = x$) for the backpropagation implementation. If you aren't familiar with calculus, this function is equivalent to the equation $y = x$. What is the slope of that equation? That is the derivative of $f(x)$.

Below, you have these tasks:

1. Implement the sigmoid function to use as the activation function. Set `self.activation_function` in `__init__` to your sigmoid function.
2. Implement the forward pass in the `train` method.
3. Implement the backpropagation algorithm in the `train` method, including calculating the output error.
4. Implement the forward pass in the `run` method.

In [17]:

```
class NeuralNetwork(object):
    def __init__(self, input_nodes, hidden_nodes, output_nodes, learning_rate):
        # Set number of nodes in input, hidden and output layers.
        self.input_nodes = input_nodes
        self.hidden_nodes = hidden_nodes
        self.output_nodes = output_nodes

        # Initialize weights
        self.weights_input_to_hidden = np.random.normal(0.0, self.hidden_nodes *
* -0.5,
                                                         (self.hidden_nodes, self
.input_nodes))

        self.weights_hidden_to_output = np.random.normal(0.0, self.output_nodes
** -0.5,
                                                          (self.output_nodes, sel
f.hidden_nodes))
        self.lr = learning_rate

        #### TODO: Set self.activation_function to your implemented sigmoid func
tion ####
        #
        # Note: in Python, you can define a function with a lambda expression,
        # as shown below.
        self.activation_function = lambda x: 1 / (1 + np.exp(-x))
        self.activation_function_prime = lambda x: x*1

        ### If the lambda code above is not something you're familiar with,
        # You can uncomment out the following three lines and put your
        # implementation there instead.
        #
        # def sigmoid(x):
        #     return 0  # Replace 0 with your sigmoid calculation here
        # self.activation_function = sigmoid

    def train(self, inputs_list, targets_list):
        # Convert inputs list to 2d array
        inputs = np.array(inputs_list, ndmin=2).T
        targets = np.array(targets_list, ndmin=2).T

        #### Implement the forward pass here ####
        ### Forward pass ###
        # TODO: Hidden layer - Replace these values with your calculations.
        # weights is (n x 56), (a set of weights for each hidden node)
        # hidden inputs linear combination & generating activated output
        hidden_inputs = np.dot(self.weights_input_to_hidden, inputs)  # signals
 into hidden layer
        hidden_outputs = self.activation_function(hidden_inputs)  # signals from
hidden layer

        # TODO: Output layer - Replace these values with your calculations.
        # values on output layer (end nodes), same steps before (linear_comb+act
ivation) using out_refs
        final_inputs = np.dot(self.weights_hidden_to_output, hidden_outputs)  #
 signals into final output layer
        final_outputs = self.activation_function_prime(final_inputs) # the deriv
ate activation function here is f(x)=x, so it is the same

        #### Implement the backward pass here ####
        ### Backward pass ###

        # TODO: Output error - Replace this value with your calculations.
        output_errors = targets_list - final_outputs  # (y - y^)Output layer err
or is the difference between desired target and actual output
        # error_term: y-y^(chapel)*f'(h), f'(h)=1, so error = error_term

        # TODO: Backpropagated error - Replace these values with your calculatio
ns.
        hidden_errors = output_errors * self.weights_hidden_to_output  # errors
 propagated to the hidden layer
        hidden_grad = hidden_outputs * (1 - hidden_outputs)  # hidden layer grad
ients

        # TODO: Update the weights - Replace these values with your calculation
s.
        # wi = wi + n(learning-rate)*error-term*xi(inputs) --> error-term
        error_term = (hidden_errors.T * hidden_grad)
        self.weights_input_to_hidden += self.lr * np.dot(error_term, inputs.T)
        # wi = wi + n(learning-rate)*output_errors*outputs_previous_layer(hidde
n) --> error-term
        self.weights_hidden_to_output += self.lr * output_errors * hidden_output
s.T


    def run(self, inputs_list):
        # Run a forward pass through the network
        inputs = np.array(inputs_list, ndmin=2).T

        #### Implement the forward pass here ####
        # TODO: Hidden layer - replace these values with the appropriate calcula
tions.
        # same from training function,but now using trained weights
        hidden_inputs = np.dot(self.weights_input_to_hidden, inputs)
        hidden_outputs = self.activation_function(hidden_inputs)

        # TODO: Output layer - Replace these values with the appropriate calcula
tions.
        final_inputs = np.dot(self.weights_hidden_to_output, hidden_outputs) # l
inear combination out_weights & hidden_out
        final_outputs = self.activation_function_prime(final_inputs)  # it will
 return final_inputs because f'(x) = x

        return final_outputs
```

In [18]:

```
def MSE(y, Y):
    return np.mean((y-Y)**2)
```

Training the network

Here you'll set the hyperparameters for the network. The strategy here is to find hyperparameters such that the error on the training set is low, but you're not overfitting to the data. If you train the network too long or have too many hidden nodes, it can become overly specific to the training set and will fail to generalize to the validation set. That is, the loss on the validation set will start increasing as the training set loss drops.

You'll also be using a method know as **Stochastic Gradient Descent (SGD) to train the network**. The idea is that for **each training** pass, you **grab a random sample of the data instead** of using the **whole data set**. You use **many more training passes than** with **normal gradient descent**, **but** each pass is **much faster**. This ends up **training the network more efficiently**. You'll learn more about SGD later.

Choose the number of epochs

This **is the number of times the dataset will pass through the network**, **each time updating** the **weights**. **As** the number of **epochs increases**, the **network becomes better and better at predicting** the targets in the training set. You'll need to **choose enough epochs to train the network well but not too many** or you'll be overfitting.

Choose the learning rate

This scales the size of weight updates. If this is too **big**, the **weights** tend to **explode** and the network fails to fit the data. A good choice to **start at is 0.1**. **If** the **network** has **problems fitting** the data, **try reducing the learning rate**. Note that the **lower** the **learning rate**, the **smaller** the **steps** are in the **weight updates** and the **longer** it takes for the **neural network to converge**.

Choose the number of hidden nodes

The **more hidden nodes** you have, the **more accurate predictions the model** will make. **Try a few different numbers** and see how it **affects the performance**. You can look at the **losses dictionary for a metric** of the network performance. If the number of hidden units is too low, then the model won't have enough space to learn and if it is too high there are too many options for the direction that the learning can take. The trick here is to find the right balance in number of hidden units you choose.

In [19]:

```
import sys

### Set the hyperparameters here ###
epochs = 1600 #16000
learning_rate = 0.000001 #0.000001
hidden_nodes = 70 # Neurons (210=overfit, 70=ok)
output_nodes = 1

N_i = train_features.shape[1]
network = NeuralNetwork(N_i, hidden_nodes, output_nodes, learning_rate)

bar = progressbar.ProgressBar(maxval=20, widgets=[progressbar.Bar('=', '[', ']'
), ' ', progressbar.Percentage()])
# bar.start()
start_time = datetime.now()
losses = {'train':[], 'validation':[]}
for e in range(epochs):
    # Go through a random batch of 128 records from the training data set
    batch = np.random.choice(train_features.index, size=5000)
    for record, target in zip(train_features.loc[batch].values,
                              train_targets.loc[batch]['QTD']):
        network.train(record, target)

    # Printing out the training progress
    train_loss = MSE(network.run(train_features), train_targets['QTD'].values)
    val_loss = MSE(network.run(val_features), val_targets['QTD'].values)
    progress = 100 * e/float(epochs)
    #bar.update(progress)
    sys.stdout.write("\rProgress: " + str(progress)[:4] \
                     + "% ... Training loss: " + str(train_loss)[:5] \
                     + " ... Validation loss: " + str(val_loss)[:5])

    losses['train'].append(train_loss)
    losses['validation'].append(val_loss)
end_time = datetime.now()
# bar.finish()
print('\nSpent Time: ', end_time - start_time)
```

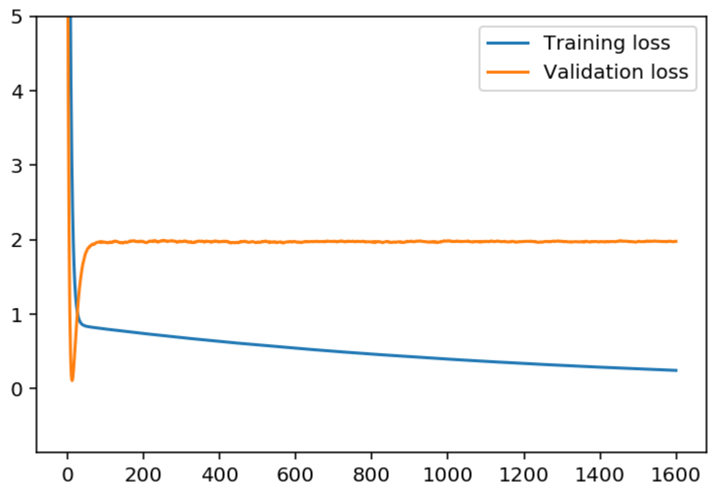
```
Progress: 99.9% ... Training loss: 0.244 ... Validation loss: 1.977
Spent Time:  0:06:07.271490
```

In [20]:

```
plt.plot(losses['train'], label='Training loss')
plt.plot(losses['validation'], label='Validation loss')
plt.legend()
plt.ylim(top=5)
# plt.ylim(top=20000, bottom=-100000)
```

Out[20]:

```
(-0.8521190607491803, 5)
```



Check out your predictions

Here, use the test data to view how well your network is modeling the data. If something is completely wrong here, make sure each step in your network is implemented correctly.

In [21]:

```
arima_targets.size
```

Out[21]:

```
616
```

In [22]:

```
test_targets.size
```

Out[22]:

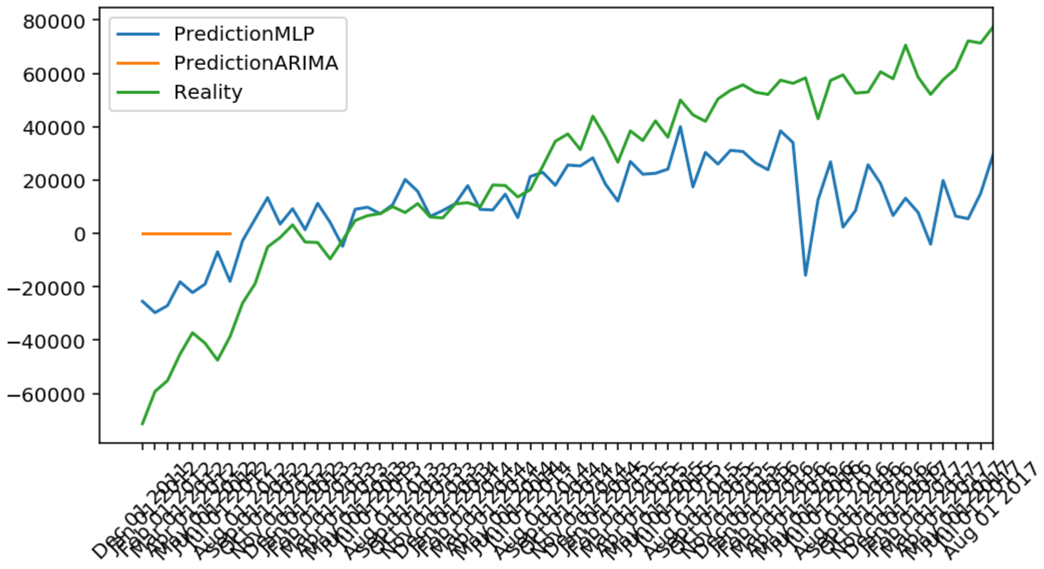
```
69
```

In [23]:

```
fig, ax = plt.subplots(figsize=(8,4))
# plt.yscale('symlog', linthreshy=1000)

mean, std = scaled_features['QTD']
predictions = network.run(test_features)*std + mean
ax.plot(predictions[0], label='PredictionMLP')
ax.plot((arima_targets['QTD']*std + mean).values, label='PredictionARIMA')
ax.plot((test_targets['QTD']*std + mean).values, label='Reality')
ax.set_xlim(right=len(predictions))
ax.legend()

dates = pd.to_datetime(microset.loc[test_data.index]['DT'])
dates = dates.apply(lambda d: d.strftime('%b %d %Y'))
ax.set_xticks(np.arange(len(dates))[:])
_ = ax.set_xticklabels(dates[:], rotation=45)
```



In [24]:

```
predictions
```

Out[24]:

```
array([[-25480.1454082 , -29715.81362803, -27069.76509658,
        -18150.39544995, -22192.53712932, -19032.29778647,
         -6912.76415508, -17985.83485097,  -2738.2332511 ,
          5375.57821441,  13394.29766836,   3466.97964678,
          9229.2003198 ,   1398.81141868,  11257.33351463,
          4146.89017245,  -4850.61185657,   9035.91023782,
          9816.18480796,   7338.67264741,  10751.57060621,
         20200.48636577,  15762.82567959,   6398.62992495,
          8546.30687993,  11134.24857349,  17892.41286704,
          8994.82698808,   8779.00674425,  14700.84703739,
          5926.8987862 ,  21351.266026  ,  22871.31972267,
         18019.35767269,  25619.60345259,  25295.2202953 ,
         28319.1149652 ,  18525.09381373,  12119.18115262,
         26947.72168778,  22176.91048834,  22516.66625684,
         24091.98820649,  40002.49248333,  17431.14446193,
         30335.33982021,  25967.32624782,  31117.20201218,
         30683.96151426,  26412.29372511,  23877.24829512,
         38438.15463492,  34029.19996723, -15700.43459979,
         12576.88360093,  26824.93587157,   2366.86530026,
          8578.18040503,  25727.42668172,  18676.82759245,
          6688.00171706,  13181.07610758,   7842.6164634 ,
         -4051.04245336,  19829.82060053,   6477.78875475,
          5500.88593265,  15004.60080876,  29576.03999808]])
```

In [25]:

```
test_features
```

Out[25]:

	MES	DT_2011-12-01 00:00:00	DT_2012-01-01 00:00:00	DT_2012-02-01 00:00:00	DT_2012-03-01 00:00:00	DT_2012-04-01 00:00:00	DT_2012-05-01 00:00:00	DT_2012-06-01 00:00:00	DT_2 (00:
0	1.629993	1	0	0	0	0	0	0	
1	-1.542224	0	1	0	0	0	0	0	
2	-1.253841	0	0	1	0	0	0	0	
3	-0.965457	0	0	0	1	0	0	0	
4	-0.677074	0	0	0	0	1	0	0	
...	...	...	...	...	...	...	...	...	
64	-0.677074	0	0	0	0	0	0	0	
65	-0.388691	0	0	0	0	0	0	0	
66	-0.100307	0	0	0	0	0	0	0	
67	0.188076	0	0	0	0	0	0	0	
68	0.476459	0	0	0	0	0	0	0	

69 rows × 70 columns

In [26]:

```
scaled_features
```

Out[26]:

```
{'MES': [6.340425531914893, 3.4498918614978353],
 'QTD': [22619.313829787236, 36022.85356023201]}
```

In [27]:

```
arima_targets['QTD']
```

Out[27]:

```
1120   -0.627916
1121   -0.627916
1122   -0.627916
1123   -0.627916
1124   -0.627916
1125   -0.627916
1126   -0.627916
1127   -0.627916
Name: QTD, dtype: float64
```

In [28]:

```
test_targets['QTD']
```

Out[28]:

```
0    -2.608054
1    -2.273090
2    -2.159926
3    -1.885316
4    -1.662006
        ...
64    0.971234
65    1.085605
66    1.375605
67    1.352671
68    1.520455
Name: QTD, Length: 69, dtype: float64
```

In [29]:

```
(arima_targets['QTD']*std + mean).values
```

Out[29]:

```
array([0., 0., 0., 0., 0., 0., 0., 0.])
```

In [30]:

```
(test_targets['QTD']*std + mean).values
```

Out[30]:

```
array([-71330.22309233, -59263.85933226, -55187.38508899, -45295.140
92532,
       -37250.8984186 , -41164.31369214, -47523.61351164, -38674.946
75425,
       -26130.27668296, -18868.71723095,  -5095.6695877 ,  -1573.595
84151,
         3198.59667261,  -3236.79733277,  -3410.72690048,  -9596.097
15227,
        -2606.30264981,   4840.0569679 ,   6644.57623292,   7427.259
28763,
        10014.46160735,   7807.730217  ,  11177.61559143,   6122.787
52978,
         5807.5401883 ,  11047.16841565,  11503.73353089,   9981.849
81341,
        18145.66889793,  17949.99813425,  13666.98252932,  16384.632
02484,
        25244.16938021,  34527.66005688,  37277.92134634,  31407.798
43603,
        43952.46850732,  36125.63796024,  26679.08831384,  38419.334
13445,
        34821.16620239,  42169.69043826,  36038.67317638,  49996.520
98534,
        44441.64541651,  41995.76087055,  50398.73311067,  53627.300
71134,
        55692.71432793,  52931.58244049,  52116.28759184,  57442.880
60304,
        56247.11482502,  58301.65784362,  42984.98528691,  57323.304
02524,
        59443.07063174,  52594.59390305,  52985.9354304 ,  60584.483
41985,
        57964.66930618,  70563.69236738,  58573.42279317,  52105.416
99385,
        57605.93957277,  61725.89620797,  72172.54086872,  71346.375
42208,
        77390.4279001 ])
```

In [31]:

```
predictions[0]
```

Out[31]:

```
array([-25480.1454082 , -29715.81362803, -27069.76509658, -18150.395
44995,
       -22192.53712932, -19032.29778647,  -6912.76415508, -17985.834
85097,
        -2738.2332511 ,   5375.57821441,  13394.29766836,   3466.979
64678,
         9229.2003198 ,   1398.81141868,  11257.33351463,   4146.890
17245,
        -4850.61185657,   9035.91023782,   9816.18480796,   7338.672
64741,
        10751.57060621,  20200.48636577,  15762.82567959,   6398.629
92495,
         8546.30687993,  11134.24857349,  17892.41286704,   8994.826
98808,
         8779.00674425,  14700.84703739,   5926.8987862 ,  21351.266
026   ,
        22871.31972267,  18019.35767269,  25619.60345259,  25295.220
2953  ,
        28319.1149652 ,  18525.09381373,  12119.18115262,  26947.721
68778,
        22176.91048834,  22516.66625684,  24091.98820649,  40002.492
48333,
        17431.14446193,  30335.33982021,  25967.32624782,  31117.202
01218,
        30683.96151426,  26412.29372511,  23877.24829512,  38438.154
63492,
        34029.19996723, -15700.43459979,  12576.88360093,  26824.935
87157,
         2366.86530026,   8578.18040503,  25727.42668172,  18676.827
59245,
         6688.00171706,  13181.07610758,   7842.6164634 ,  -4051.042
45336,
        19829.82060053,   6477.78875475,   5500.88593265,  15004.600
80876,
        29576.03999808])
```

Result:

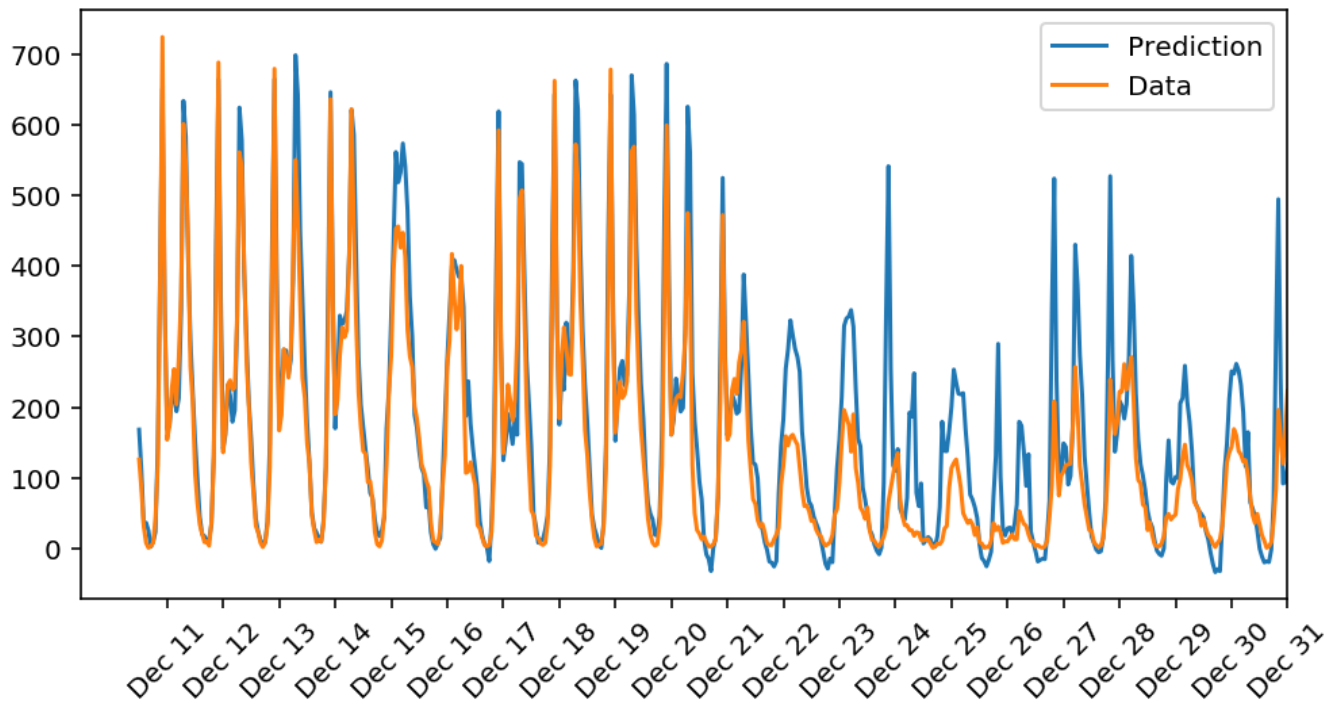
Considering 60% of safety range the **BNAFARIUNS** predicted 395,2(247.55777106*1,6); 377,6(236.28421505*1,6). Meaning just 7 extra medicines (from 388 (real) to 395 (predicted)) for for 2018-10 and 101 medicines for 2018-11. **Question**: Was it model prediction a good one?

> **Note:** The time frame was too short, creating fictional data for more months increased the assertivity.

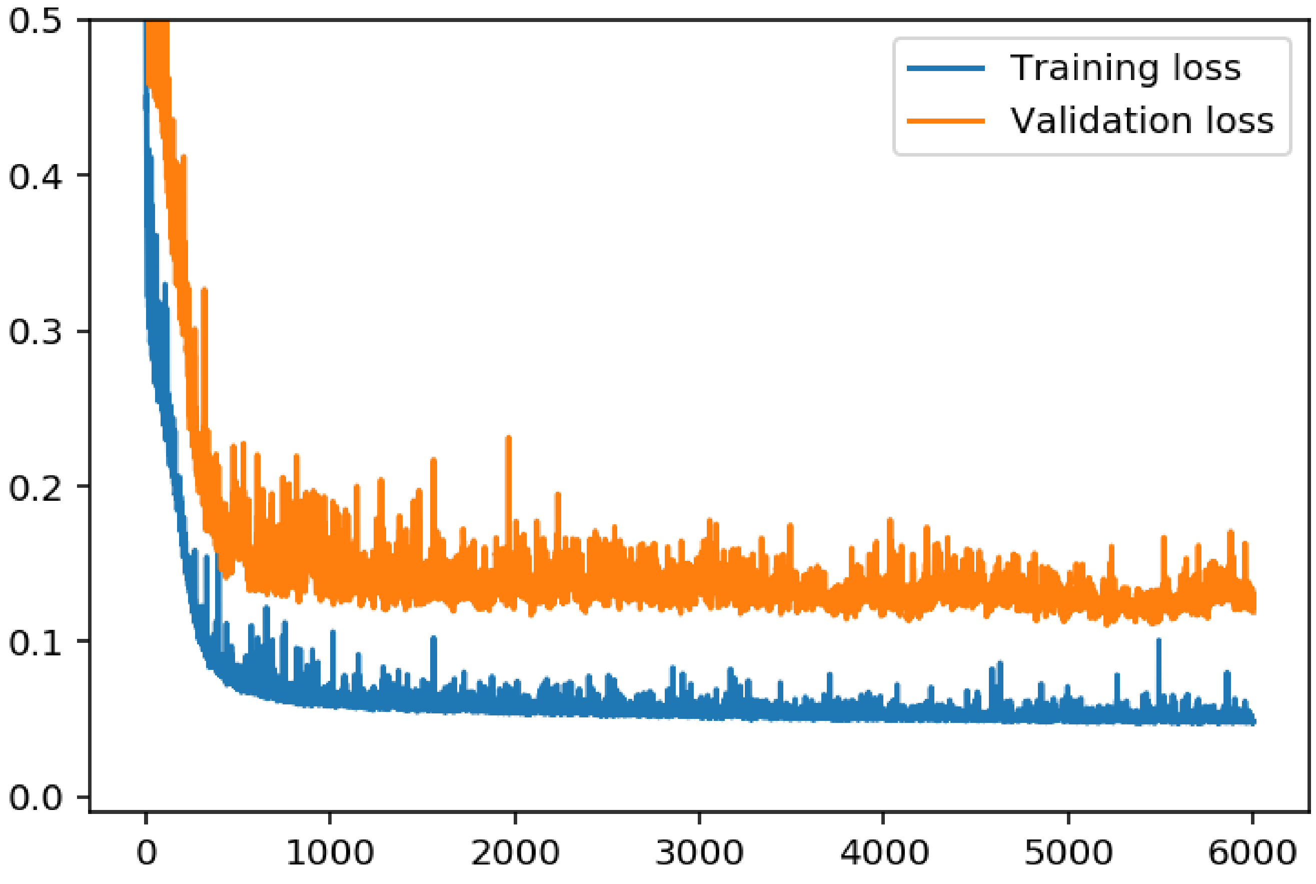
Answer

DAF Resp:

Sample with better data



Training with better data (loss decay)


Training loss
Validation loss