


Branch: master ▾

ai-notebooks / Corporación+Favorita+Grocery+Sales+Forecasting.ipynb

Find file

Copy path

 **kevindewalt** Rename Shared+on+Kaggle.ipynb to Corporación+Favorita+Grocery+Sales+F... c7b34bb 3 days ago

1 contributor

1713 lines (1712 sloc) 53.2 KB

## Simple 4-feature Neural Network for LB .547

A few months ago I started trying to apply the techniques Jeremy Howard shares [here](https://github.com/fastai/courses/blob/master/deeplearning2/rossman.ipynb) (<https://github.com/fastai/courses/blob/master/deeplearning2/rossman.ipynb>) for making predictions using simple feed-forward neural networks and embedding layers to this contest.

It is based on the [this paper](https://arxiv.org/abs/1604.06737) (<https://arxiv.org/abs/1604.06737>) by Guo and Berkhahn who finished 3rd place in the [Rossman Store Sales](https://www.kaggle.com/c/rossmann-store-sales) (<https://www.kaggle.com/c/rossmann-store-sales>) contest.

Well ... it didn't work very well. Here is a quick summary of what I tried along with a simple approach which got me the best results.

**Feel free to ask questions or share any advice in the comments.**

I'm presuming a lot of knowledge about neural networks. If you take the time to carefully read through my results I'll be happy to share what I know.

### What didn't work - a massive, feature-rich dataset

I started by systematically building features with all data provided in the contest following the techniques Jeremy Howard outlines in the link above.

I generated over 27 category features (e.g. year, month, dayofweek, family, holidays) and 15 continuous features (e.g. oil price, 30-day oil change, days before/after promotion).

Some of these were quite complex such as days before/after a promotion given that each depends on a store and item.

And I did this for **all of the training data**, resulting in dataframes > 20 GB.

Then I built a sophisticated embedding layer and a few dense layers for my model. It took several hours to train on a GTX1080Ti ... and my results were pretty lousy.

While it's always possible I made a mistake, I couldn't get anything below 1.0 on the public leaderboard. So I decided to scrap it and try another approach.

### What worked better - 4 features and a few months of training data

I decided to start afresh trying replicate the results using a handful of features and less data.

I started with the 'moving average' technique presented by Paulo Pinto and others [here](https://www.kaggle.com/paulorzp/log-ma-and-days-of-week-means-lb-0-529/code) (<https://www.kaggle.com/paulorzp/log-ma-and-days-of-week-means-lb-0-529/code>) and tuned the neural network to get a LB score of .547. The 4 features are:

- store number
- item number
- whether or the item was on promotion
- an average of recent sales at the same store/item combinations.

I used some unorthodox techniques (no dropout, batch sizes of 500,000) which I shared below.

I got better results and after trying to add lots of additional features never improved on these results.

### What I would like to try (but probably won't)

This contest was a good reminder to start with simple models, small amounts of training data and gradually add features.

My goal was to learn the techniques presented in the Guo paper for using embedding layers on time-series data. I definitely achieved it and now have a good understanding of the approach.

#### more features

I'd be curious to see if adding oil, weather, holiday, geographical etc. information improves results but I'm not super-optimistic about getting a big boost.

#### custom sample weight and loss functions

On the model side I'd like to figure out why using Keras sample\_weight field isn't working and would like to build a custom loss functions with the NWRMSLE instead of MAE. Unfortunately building custom loss functions in Keras gets quite complex using the Keras backend. Keras is better for out-of-the-box functions.

#### A good reminder to start simple

Unfortunately, most of the data in this contest doesn't appear very predictive. In this case a simpler algorithm like xgboost is probably a better

Unfortunately most of the data in this contest doesn't appear very predictive. In this case a simpler algorithm like xgboost is probably a better choice.

Obviously ... we would never tackle a client project using this approach. Start simple, learn the data, and get a good baseline is always the right approach.

Hope it helps!

## Load libraries

```
In [19]: #Select between 2 GTX1080Ti GPUs
gpu=1
%env CUDA_DEVICE_ORDER=PCI_BUS_ID
%env CUDA_VISIBLE_DEVICES=$gpu
# Libraries and utilities
%matplotlib inline
import importlib
#file kevin has most of the standard libraries: numpy, pandas, keras ...
import kevin; importlib.reload(kevin)
from kevin import *
from datetime import timedelta
from sklearn_pandas import DataFrameMapper
from sklearn.preprocessing import LabelEncoder, Imputer, StandardScaler

env: CUDA_DEVICE_ORDER=PCI_BUS_ID
env: CUDA_VISIBLE_DEVICES=1
kevin loaded
```

## Functions

```
In [2]: #paths
slowdata = data_path + 'slowdisk/'; slowdata
simple = slowdata + 'simple2/'
#Leaves duplicate field names on left and appends with _y on right
def join_df(left, right, left_on, right_on=None):
    if right_on is None: right_on=left_on
    return left.merge(right, how='left', left_on=left_on, right_on=right_on, suffixes=("", "_y"))
```

## Load and clean data

```
In [3]: # specify datatypes before loading the data will save you a ton of memory.
dtypes = {'id': np.uint32,
          'store_nbr': np.uint8,
          'item_nbr': np.uint32,
          'unit_sales': np.float32,
          'class': np.uint16,
          'dcoilwtico': np.float16,
          'transactions': np.uint16,
          'cluster': np.uint32}
```

```
In [4]: train = pd.read_csv(slowdata + 'train.csv', dtype=dtypes, usecols=[1,2,3,4,5], parse_dates=['date'],
                           skiprows=range(1, 86672217) #Skip dates before 2016-08-01, and id column
                           )
test = pd.read_csv(slowdata + 'test.csv', dtype=dtypes, parse_dates=['date'])
```

```
In [5]: # data older than march 1 showed no improvement
train = train[train['date'] >= '2017-03-01']
```

```
In [6]: train['unit_sales'] = train['unit_sales'].as_matrix().clip(min=0) # get rid of negs
train['unit_sales'] = train['unit_sales'].apply(pd.np.log1p) # transform for smoother training
train['onpromotion'].fillna(False, inplace=True) # fill in the missing promotion data with 'False'
test['onpromotion'].fillna(False, inplace=True)
```

```
In [7]: #add 'day of the week' to data
train['dow'] = train['date'].dt.dayofweek.astype(np.int8)
test['dow'] = test['date'].dt.dayofweek.astype(np.int8)
```

## Save and load cleaned data

```
In [8]: train.to_pickle(simple + 'train_processed.pkl')
```

```
test.to_pickle(simple + 'test_processed.pkl')
```

```
In [9]: train = pd.read_pickle(simple + 'train_processed.pkl')
test = pd.read_pickle(simple + 'test_processed.pkl')
```

## Derive day-of-the-week dataframes

This is technique for deriving the key feature for the training algorithm. Before diving into the code consider an example:

### Example

Suppose you want to estimate the sale of Beer at your local 7-11.

You could just calculate the average amount of (recent) daily sales of Beer at the 7-11. For instance, we may know that 7-11 sells an average of 10 cases a beer every day. So we could just use '10' as the prediction for beer sales.

But as it turns out Beer doesn't sell the same every day. Customers buy more Beer on Friday and Saturday and less on Sunday.

madw/mawk corrects for this daily factor. After applying this factor we submit a score which estimates 8 for Tues-Thurs, 4 for Sunday, 21 for Friday and Saturday. Average is still 10 but we've scaled for the store and item number.

### Confused? Calculate for 1 store

I was initially confused by this technique. After I calculate the same results for 1 store it was clearer.

### Not really a "moving average"

I kept the syntax for readability, but this isn't a 'moving' average. It is a daily average by store and item.

```
In [10]: # from https://www.kaggle.com/paulorzp/log-ma-and-days-of-week-means-lb-0-529/code and others
ma_dw = train[['item_nbr', 'store_nbr', 'dow', 'unit_sales']].groupby(
    ['item_nbr', 'store_nbr', 'dow'])['unit_sales'].mean().to_frame('madw').reset_index()
ma_wk = ma_dw[['item_nbr', 'store_nbr', 'madw']].groupby(
    ['item_nbr', 'store_nbr'])['madw'].mean().to_frame('mawk').reset_index()
```

```
In [13]: #remember the madw and mawk sales have been transformed by log1p above
ma_dw.head()
```

```
Out[13]:
```

	item_nbr	store_nbr	dow	madw
0	96995	1	0	0.895880
1	96995	1	1	0.693147
2	96995	1	2	0.693147
3	96995	1	3	0.794513
4	96995	1	4	0.943827

```
In [14]: ma_wk.head()
```

```
Out[14]:
```

	item_nbr	store_nbr	mawk
0	96995	1	0.844024
1	96995	2	0.775311
2	96995	3	0.786271
3	96995	4	0.783251
4	96995	5	0.882478

## Create multi-index

```
In [15]: # Expanding the training data to list a store/item combination on every data and filling zeros
# where no data is listed.
# I'm not totally comfortable with this approach because it presumes missing data means no sales.
# however, I did get better results by doing it, so I continued.
```

```

u_dates = train.date.unique()
u_stores = train.store_nbr.unique()
u_items = train.item_nbr.unique()
train.set_index(['date', 'store_nbr', 'item_nbr'], inplace=True)
train=train.reindex(pd.MultiIndex.from_product((u_dates, u_stores, u_items),
names = ['date', 'store_nbr', 'item_nbr'])).reset_index()
train.unit_sales.fillna(0, inplace=True)
train.onpromotion.fillna(0, inplace=True)
del u_dates, u_stores, u_items # save memory

```

## 'Moving' averages

More correctly stated as static revent averages over a time window.

```

In [16]: lastdate = train.iloc[train.shape[0]-1].date
# Only necessary to create dataframe for the loop below. We copy over 'mais' with median later.
ma_is = train[['item_nbr', 'store_nbr', 'unit_sales']].groupby(
    ['item_nbr', 'store_nbr'])['unit_sales'].mean().to_frame('mais')

```

```
In [17]: ma_is.head()
```

```
Out[17]:
```

		mais
item_nbr	store_nbr	
96995	1	0.128293
	2	0.156473
	3	0.216649
	4	0.062589
	5	0.092171

```

In [20]: # Now calculate average sales of store/item combinations over recent time windows
# based on days before the last available training data date.
# e.g. 112 days before the last date, 56 before the last date ...
for i in [112, 56, 28, 14, 7, 3, 1]:
    tmp = train[train.date>lastdate-timedelta(int(i))]
    tmpg = tmp.groupby(['item_nbr', 'store_nbr'])['unit_sales'].mean().to_frame('mais'+str(i))
    ma_is = ma_is.join(tmpg, how='left')

```

```
In [21]: del tmp; del tmpg
```

```

In [22]: # Now take the median of of the day windows we calculated.
# I tried just keeping all of the windows but got worse results.
# Perhaps mean would work as well as median - I didn't test it.
# In any case, this is a pretty simplistic feature which may not generalize.
ma_is['mais'] = ma_is.median(axis=1)

```

```
In [23]: ma_is.head()
```

```
Out[23]:
```

		mais	mais112	mais56	mais28	mais14	mais7	mais3	mais1
item_nbr	store_nbr								
96995	1	0.141274	0.154255	0.172356	0.295202	0.334438	0.099021	0.000000	0.000000
	2	0.024755	0.161961	0.123776	0.049511	0.000000	0.000000	0.000000	0.000000
	3	0.355917	0.208903	0.286789	0.336299	0.375535	0.454008	0.462098	0.693147
	4	0.124828	0.093884	0.150635	0.099021	0.099021	0.198042	0.231049	0.693147
	5	0.118639	0.138257	0.202249	0.237278	0.099021	0.198042	0.000000	0.000000

```

In [25]: # Now remove the mais112 ... mais1 and merge the mais feature with training dataframe
ma_is.reset_index(inplace=True)
ma_is.drop(list(ma_is.columns.values)[3:], 1, inplace=True)
train=join_df(train, ma_is, ['item_nbr', 'store_nbr'])

```

```
In [26]: ma_is.head()
```

```
Out[26]:
```

Out[20]:

	item_nbr	store_nbr	mais
0	96995	1	0.141274
1	96995	2	0.024755
2	96995	3	0.355917
3	96995	4	0.124828
4	96995	5	0.118639

## Scale the average sales by dow

```
In [27]: # now you see the purpose of 'madw' and 'mawk'. To scale each sales average by
# how well it historically sells for that day of the week
train = pd.merge(train, ma_wk, how='left', on=['item_nbr', 'store_nbr'])
train = pd.merge(train, ma_dw, how='left', on=['item_nbr', 'store_nbr', 'dow'])
train['m_ratio'] = train['mais'] * train['madw'] / train['mawk']
```

## m\_ratio is the key feature

Lots of work and data wrangling to get to that point.

```
In [28]: # yikes, this is big. Let's clean it up.
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 36278928 entries, 0 to 36278927
Data columns (total 10 columns):
date           datetime64[ns]
store_nbr      uint64
item_nbr       uint64
unit_sales     float32
onpromotion    object
dow            object
mais           float64
mawk           float32
madw           float32
m_ratio        float64
dtypes: datetime64[ns](1), float32(3), float64(2), object(2), uint64(2)
memory usage: 2.6+ GB
```

```
In [29]: train['store_nbr'] = train['store_nbr'].astype(np.uint8)
train['onpromotion'] = train['onpromotion'].astype(np.uint8)
train['item_nbr'] = train['item_nbr'].astype(np.uint32)
# I reversed the sales normalization here. I apply a different
# normalization function before training below.
# I didn't test but expect both to give similar results.
train['unit_sales'] = train['unit_sales'].apply(np.expml)
train.drop('dow', 1, inplace=True)
train.drop('mawk', 1, inplace=True)
train.drop('madw', 1, inplace=True)
train.drop('mais', 1, inplace=True)
```

```
In [30]: # Now smaller
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 36278928 entries, 0 to 36278927
Data columns (total 6 columns):
date           datetime64[ns]
store_nbr      uint8
item_nbr       uint32
unit_sales     float32
onpromotion    uint8
m_ratio        float64
dtypes: datetime64[ns](1), float32(1), float64(1), uint32(1), uint8(2)
memory usage: 1.1 GB
```

## Now prep the test data

```
In [31]: test = pd.merge(test, ma_is, how='left', on=['item_nbr', 'store_nbr'])
test = pd.merge(test, ma_wk, how='left', on=['item_nbr', 'store_nbr'])
test = pd.merge(test, ma_dw, how='left', on=['item_nbr', 'store_nbr', 'dow'])
```

```
test = pd.merge(test, ma_dw, how=left, on=[ 'item_nbr', 'store_nbr', 'dow'])
test['m_ratio'] = test['mais']*test['madw']/test['mawk']
test['store_nbr'] = test['store_nbr'].astype(np.uint8)
test['onpromotion'] = test['onpromotion'].astype(np.uint8)
test['item_nbr'] = test['item_nbr'].astype(np.uint32)
test.drop('id', 1, inplace=True)
test.drop('dow', 1, inplace=True)
test.drop('mawk', 1, inplace=True)
test.drop('madw', 1, inplace=True)
test.drop('mais', 1, inplace=True)
```

In [32]: `del ma_is; del ma_wk; del ma_dw`

In [33]: `# clean up and rescale m_ratio, applying the same denormalization we did to unit sales.  
# probably makes no difference but didn't test it.  
for m in ['m_ratio']:  
 train[m].fillna(0, inplace=True)  
 test[m].fillna(0, inplace=True)  
 train[m] = train[m].apply(np.exp1) # probably unnecessary  
 test[m] = test[m].apply(np.exp1)`

## Create NN features

We now need to scale the continuous variables (in this case there is only one, `m_ratio`) and turn the category variables to discrete categories.

I'm following the technique demonstrated by Jeremy Howard from the Rossman contest, <https://github.com/fastai/courses/blob/master/deeplearning2/rossman.ipynb> (<https://github.com/fastai/courses/blob/master/deeplearning2/rossman.ipynb>)

However ... this is overkill for such a small number of features.

### Category

In [34]: `#used to make the embedding layer.  
cat_var_dict = {'item_nbr': 5,  
 'onpromotion': 3,  
 'store_nbr': 3,  
 }  
# time-saver if you have a lot of categories. Overkill here.  
cat_vars = [o[0] for o in  
 sorted(cat_var_dict.items(), key=lambda x: x[1], reverse=True)]`

In [35]: `cat_vars`

Out[35]: `['item_nbr', 'onpromotion', 'store_nbr']`

### Continuous

In [36]: `# just 1!  
contin_vars = ['m_ratio']`

## Apply LabelEncoder and StandardScaler with DataFrameMapper

Again, overkill for this small example but this code really simplifies when you have dozens of features.

In [37]: `# DataFrameMapper takes a list of dataframe columns and sklearn transformations.  
# Seems a bit weird but check the DataFrameMapper docs.  
# First create a list for the categories and columns.  
cat_maps = [(o, LabelEncoder()) for o in cat_vars]  
contin_maps = [(o, StandardScaler()) for o in contin_vars]  
#Now use DataFrameMapper to apply LabelEncoder to the category variables and StandardScaler  
#to the continuous ones.  
cat_mapper = DataFrameMapper(cat_maps)  
contin_mapper = DataFrameMapper(contin_maps)  
# Now call the fit function in DataFrameMapper  
# need to append the test data to make sure we account for items, stores not in training data.  
cat_map_fit = cat_mapper.fit(train.append(test))  
contin_map_fit = contin_mapper.fit(train.append(test))`

*# we can now apply the DataFrameMapper.transform function to transform the data*

```
#we can now apply the DataTransformer transform function to transform the data.
#we do this in the cat_preproc and contin_preproc functions below.
```

## Split train/valid

```
In [38]: #use the last 10% of data for validation.
train_ratio = 0.90
train_size = int(train_ratio*len(train))
trn = train[:train_size]
vld = train[train_size:]
```

## Preprocess

```
In [39]: #functions to use to make the category and continuous variables
def cat_preproc(dat):
    return cat_map_fit.transform(dat).astype(np.int16)
def contin_preproc(dat):
    return contin_map_fit.transform(dat).astype(np.float32)
```

```
In [40]: # now make the input features
cat_map_train = cat_preproc(trn)
cat_map_valid = cat_preproc(vld)
cat_map_test = cat_preproc(test)

contin_map_train = contin_preproc(trn)
contin_map_valid = contin_preproc(vld)
contin_map_test = contin_preproc(test)

#for use later
contin_cols=contin_map_train.shape[1]
```

## Save and load data

```
In [41]: #save the model data
pickle.dump(contin_map_fit, open(simple + 'contin_maps.pickle', 'wb'))
pickle.dump(cat_map_fit, open(simple + 'cat_maps.pickle', 'wb'))
pickle.dump(cat_var_dict, open(simple + 'cat_var_dict.pkl', 'wb'))
pickle.dump(cat_map_train, open(simple + 'cat_map_train.pkl', 'wb'), protocol=4)
pickle.dump(cat_map_valid, open(simple + 'cat_map_valid.pkl', 'wb'))
pickle.dump(contin_map_train, open(simple + 'contin_map_train.pkl', 'wb'), protocol=4)
pickle.dump(contin_map_valid, open(simple + 'contin_map_valid.pkl', 'wb'))
pickle.dump(cat_map_test, open(simple + 'cat_map_test.pkl', 'wb'))
pickle.dump(contin_map_test, open(simple + 'contin_map_test.pkl', 'wb'))
pickle.dump(contin_cols, open(simple + 'contin_cols.pkl', 'wb'))
trn.to_pickle(simple + 'trn.pkl')
vld.to_pickle(simple + 'vld.pkl')
```

```
In [42]: cat_map_train = pickle.load(open(simple + 'cat_map_train.pkl', 'rb'))
cat_map_valid = pickle.load(open(simple + 'cat_map_valid.pkl', 'rb'))
contin_map_train = pickle.load(open(simple + 'contin_map_train.pkl', 'rb'))
contin_map_valid = pickle.load(open(simple + 'contin_map_valid.pkl', 'rb'))
cat_map_fit = pickle.load(open(simple + 'cat_maps.pickle', 'rb'))
contin_map_fit = pickle.load(open(simple + 'contin_maps.pickle', 'rb'))
cat_var_dict = pickle.load(open(simple + 'cat_var_dict.pkl', 'rb'))
cat_map_test = pickle.load(open(simple + 'cat_map_test.pkl', 'rb'))
contin_map_test = pickle.load(open(simple + 'contin_map_test.pkl', 'rb'))
contin_cols = pickle.load(open(simple + 'contin_cols.pkl', 'rb'))
trn = pd.read_pickle(simple + 'trn.pkl')
vld = pd.read_pickle(simple + 'vld.pkl')
```

## Create model data

```
In [43]: #List of (rows,1) arrays for training
def split_cols(arr): return np.hsplit(arr,arr.shape[1])
```

```
In [44]: #just some simple array manipulations to prep it for the model.
map_train = split_cols(cat_map_train) + [contin_map_train]
map_valid = split_cols(cat_map_valid) + [contin_map_valid]
map_test = split_cols(cat_map_test) + [contin_map_test]
```



## Normalizing function

Normalizing the labels

```
In [45]: def normy(y):
        meany=np.mean(y)
        stdevy = np.std(y)
        ynorm = (y-meany)/stdevy
        return meany, stdevy, ynorm

        def denorm(y, meany, stdevy, r=3):
            return np.round(y*stdevy+mean, r)
```

## Sample weighting

```
In [46]: #weighting samples using keras sample_weight made results worse.
        sample_weight = None
```

## Load y

```
In [47]: #mean_ and stdevy_ used for denorm function
        meany_train, stdevy_train, y_train = normy(trn.as_matrix(columns = ['unit_sales']))
        meany_valid, stdevy_valid, y_valid = normy(vld.as_matrix(columns = ['unit_sales']))
```

## Create Model

```
In [48]: #generates embedding layer for categorical variables
        #returns input tensor for creating model with Keras model function
        def get_emb(name, n_in, n_out, reg, shape=1):
            inp = Input(shape=(shape,), name=name+'_in')
            em = Embedding(n_in, n_out, input_length=shape, W_regularizer=l2(reg))(inp)
            em = Flatten(name=name+'_flt')(em)
            return inp, em
```

```
In [49]: #helper functions
        def cat_input_dim(feats): return len(feats[1].classes_)
        def cat_input_name(feats): return feats[0]
```

```
In [50]: #continuous input
        contin_inp = Input((contin_cols,), name='contin')
        contin_dense = Dense(contin_cols*2, activation='relu', name='contin_dense')(contin_inp)

        #category embeddings
        embs = [get_emb(name=cat_input_name(feats), n_in=cat_input_dim(feats),
                        n_out=cat_var_dict[cat_input_name(feats)], reg=1e-4)
                for feats in cat_map_fit.features]

        x = merge([em for _, em in embs] + [contin_dense], mode='concat')
        # x = Dropout(0.05)(x) with so few Dense layers dropout was unnecessary
        x = Dense(10, activation='relu', init='uniform')(x)
        x = BatchNormalization()(x)
        # x = Dropout(0.1)(x)
        x = Dense(5, activation='relu', init='uniform')(x)
        x = BatchNormalization()(x)
        # x = Dropout(0.1)(x)
        x = Dense(1, activation='linear')(x)
        model = Model([inp for inp, _ in embs] + [contin_inp], x)
        lr = 1e-1
        model.compile(optimizer=Adam(lr=lr), loss='mean_absolute_error')
```

```
In [51]: #note that most of the features are in the item number embedding layer - an unfortunate
        #consequence of having so few features.
        model.summary()
```

Layer (type)	Output Shape	Param #	Connected to
item_nbr_in (InputLayer)	(None, 1)	0	

onpromotion_in (InputLayer)	(None, 1)	0	
store_nbr_in (InputLayer)	(None, 1)	0	
embedding_1 (Embedding)	(None, 1, 5)	20295	item_nbr_in[0][0]
embedding_2 (Embedding)	(None, 1, 3)	6	onpromotion_in[0][0]
embedding_3 (Embedding)	(None, 1, 3)	162	store_nbr_in[0][0]
contin (InputLayer)	(None, 1)	0	
item_nbr_flt (Flatten)	(None, 5)	0	embedding_1[0][0]
onpromotion_flt (Flatten)	(None, 3)	0	embedding_2[0][0]
store_nbr_flt (Flatten)	(None, 3)	0	embedding_3[0][0]
contin_dense (Dense)	(None, 2)	4	contin[0][0]
merge_1 (Merge)	(None, 13)	0	item_nbr_flt[0][0] onpromotion_flt[0][0] store_nbr_flt[0][0] contin_dense[0][0]
dense_1 (Dense)	(None, 10)	140	merge_1[0][0]
batchnormalization_1 (BatchNormaliza	(None, 10)	40	dense_1[0][0]
dense_2 (Dense)	(None, 5)	55	batchnormalization_1[0][0]
batchnormalization_2 (BatchNormaliza	(None, 5)	20	dense_2[0][0]
dense_3 (Dense)	(None, 1)	6	batchnormalization_2[0][0]
=====			
Total params: 20,728			
Trainable params: 20,698			
Non-trainable params: 30			

## Training

```
In [52]: #function to allow me easier retraining to test model development and change Learning
#rate on the fly
def retrain(model=model, map_train=map_train, y_train=y_train, sample_weight=sample_weight, map_valid=map_
valid, y_valid=y_valid,
          epochs=[10,10,10], learns = [1e-2, 1e-3, 1e-4], batch_size=5096):
    hist=[]
    loops = len(epochs)
    if len(learns) != loops:
        raise ValueError('number of epochs much match number of learning rates')
    for l in range(loops):
        K.set_value(model.optimizer.lr, learns[l])
        history = model.fit(map_train, y_train, batch_size=batch_size, sample_weight=sample_weight,
                           nb_epoch=epochs[l], validation_data=(map_valid, y_valid))
        hist.append(history.history)
    return hist
```

```
In [53]: #yes ... this looks like an insanely high batch size.
#each training example is very small.
#obviously can't do this with images!
#The model trained faster and was model stable with large batches.
model.fit(map_train, y_train, batch_size=500000, nb_epoch=1, validation_data=(map_valid, y_valid))
```

```
Train on 32651035 samples, validate on 3627893 samples
Epoch 1/1
32651035/32651035 [=====] - 20s - loss: 0.1615 - val_loss: 1.4143
```

```
Out[53]: <keras.callbacks.History at 0x7f090b48eeb8>
```

```
In [54]: #2 more training cycles with Lower Learning rate
hist = retrain(epochs=[1,1], learns = [1e-2, 1e-3], batch_size=100000)
```

```
Train on 32651035 samples, validate on 3627893 samples
Epoch 1/1
32651035/32651035 [=====] - 22s - loss: 0.1121 - val_loss: 0.1338
Train on 32651035 samples, validate on 3627893 samples
Epoch 1/1
```

 1/1

```
32651035/32651035 [=====] - 23s - loss: 0.1093 - val_loss: 0.1092
```