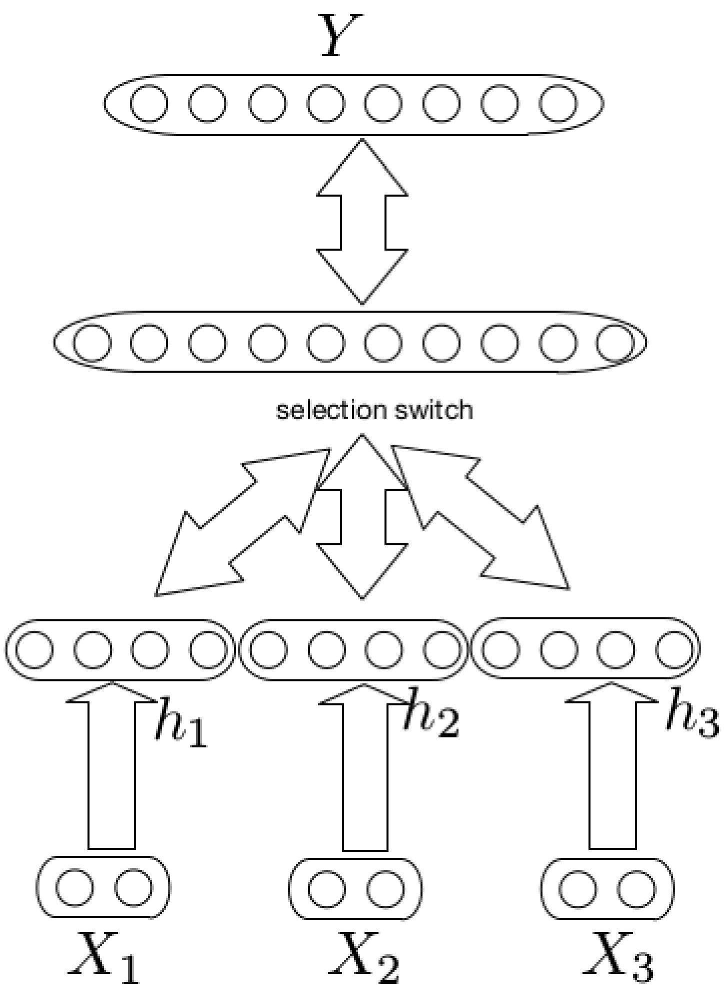
ike 👍,
but humans create 🎨, discover 🚀 and love ❤️
Jul 9, 2017 · 10 min read

Neural networks for algorithmic trading. Multimodal and multitask deep learning



Almost multimodal learning model

Here we are again! We already have four tutorials on financial forecasting with artificial neural networks where we compared different architectures for financial time series forecasting, realized how to do this forecasting adequately with correct data preprocessing and regularization, performed our forecasts based on multivariate time series and could produce really nice results for volatility forecasting and implemented custom loss functions. For solving every of latter problems we used a individual model trained on particular data: we always had we had one input and one output. But we also know, that neural networks are actually computational graphs where we can pass different data in and have several outputs as well. And it's very suitable for our problem, where we want to have inputs of different nature (today we will try to merge time series and text data) and forecast different things based on a single neural network.

Previous posts:

1. Simple time series forecasting (and mistakes done)
2. Correct 1D time series forecasting + backtesting
3. Multivariate time series forecasting
4. Volatility forecasting and custom losses
5. Multitask and multimodal learning
6. Hyperparameters optimization
7. Enhancing classical strategies with neural nets
8. Probabilistic programming and Pyro forecasts

As always, code is available on the Github.

Daily News for Stock Market Prediction dataset

In this tutorial we will use dataset, that contains not only multivariate time series, but also text data with daily news corresponding to trading days from Kaggle. You can check the details of the dataset on the link before, here is short summary what is inside:

> ***News data**: I crawled historical news headlines from Reddit WorldNews Channel (/r/worldnews). They are ranked by reddit users' votes, and only the top 25 headlines are considered for a single date. (Range: 2008–06–08 to 2016–07–01)*

> ***Stock data**: Dow Jones Industrial Average (DJIA) is used to "prove the concept". (Range: 2008–08–08 to 2016–07–01)*

I prepared a script for loading all data in useful for us form, so we will not dive into data loading in details. Workflow with time series is like in all tutorials before, some details of text preparation will be discussed later. It's only worth to mention in advance, that now for every day we have two vectors—first is our usual OHLCV tuple, second one is vector, obtained from the news data (yes, we gonna use word2vec). But let's stay for a while with our candles :)

For those, who gonna check the code, I want to clarify variables:

```
X_train # Time series data
X_train_text # word2vec decoded text data
Y_train # Labels for voaltility
Y_train2 # Labels for classification (movement direction)
```

Multitask learning

In first part of our tutorial we will research multitask learning. Wikipedia says:

> ***Multi-task learning** (MTL) is a subfield of machine learning in which multiple learning tasks are solved at the same time, while exploiting commonalities and differences across tasks. This can result in improved learning efficiency and prediction accuracy for the task-specific models, when compared to training the models separately.*

In our case I am very curious, if we want to predict, for example, volatility (it worked well before), how can we help our network to perform better adding additional information to the loss about, for example, movement direction? Adding auxiliary loss function can help neural network to learn different representation, based not only on variability of time series, but also on movement direction.

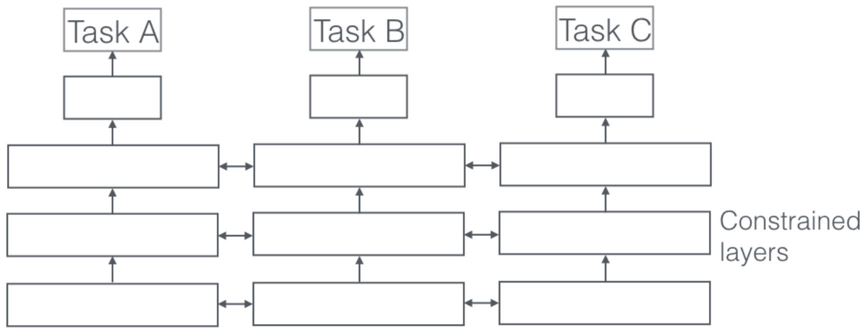


Illustration from http://ruder.io/

Looking on the picture above idea is more clear—we train one set of layers of a neural architecture to solve several tasks, and while backpropagation errors of all of them will be propagated through shared layers.

To understand multitask learning (MTL) better, I suggest you to read

1. This blog post of Sebastian Ruder
2. Representation learning chapter of famous Deep Learning book by Bengio, Goodfellow et al.

Let's perform first simple stacked LSTMs to forecast volatility as continuous number. Here is network for it:

```
main_input = Input(shape=(30, 5), name='ts_input')

lstm1 = LSTM(10, return_sequences=True,
recurrent_dropout=0.25, dropout=0.25,
bias_initializer='ones')(main_input)
lstm1 = LSTM(10, return_sequences=True,
recurrent_dropout=0.25, dropout=0.25,
bias_initializer='ones')(lstm1)
lstm1 = Flatten()(lstm1)

x = Dense(64)(lstm1)
x = LeakyReLU()(x)
x = Dense(1, activation = 'linear')(x)

final_model = Model(inputs=[main_input], outputs=[x])
```

```
opt = Nadam(lr=0.002, clipnorm = 0.5)

reduce_lr = ReduceLROnPlateau(monitor='val_loss',
factor=0.9, patience=50, min_lr=0.000001, verbose=1)

checkpointer = ModelCheckpoint(monitor='val_loss',
filepath="model.hdf5", verbose=1, save_best_only=True)

final_model.compile(optimizer=opt,
              loss='mse')
```

And train the network with this code:

```
history = final_model.fit(X_train, Y_train,
              nb_epoch = 100,
              batch_size = 256,
              verbose=1,
              validation_data=(X_test, Y_test),
              callbacks=[reduce_lr, checkpointer],
              shuffle=True)
```

The loss function evolution during the training looks like:

Loss function for 2-layer RNN volatility forecast

And prediction on test data looks like:

Prediction of volatility of 2-layer RNN

Looks not bad at all, we could capture main dependencies and can predict biggest jumps. The **MSE is 0.0161, MAE 0.073 is and MAPE is 3.01%**. But let's check if we can do it better!

Now we are coming to multitask learning. It's really easier to implement than to understand, especially with Keras functional API. We just add new "branch" of a network, call it *x2* and set over it additional output. After we need to add it to final model and set a loss in *compile* function. One very important moment is, that I emphasize the attention of the model on volatility forecasting, so I set weight for binary crossentropy loss 0.2. Another reason to do it is that our MSE loss will be much smaller during the training and we need to react on it's changes more.

```
main_input = Input(shape=(30, 5), name='ts_input')

lstm1 = LSTM(10, return_sequences=True,
recurrent_dropout=0.25, dropout=0.25,
bias_initializer='ones')(main_input)
lstm1 = LSTM(10, return_sequences=True,
recurrent_dropout=0.25, dropout=0.25,
bias_initializer='ones')(lstm1)
lstm1 = Flatten()(lstm1)

x1 = Dense(64)(lstm1)
x1 = LeakyReLU()(x1)
x1 = Dense(1, activation = 'linear', name='regression')(x1)

x2 = Dense(64)(lstm1)
x2 = LeakyReLU()(x2)
```

```
x2 = Dropout(0.9)(x2)
x2 = Dense(1, activation = 'sigmoid', name = 'class')(x2)

final_model = Model(inputs=[main_input],
              outputs=[x1, x2])

opt = Nadam(lr=0.002, clipnorm = 0.5)

reduce_lr = ReduceLROnPlateau(monitor='val_loss',
factor=0.9, patience=50, min_lr=0.000001, verbose=1)
checkpointer = ModelCheckpoint(monitor='val_loss',
filepath="model.hdf5", verbose=1, save_best_only=True)

final_model.compile(optimizer=opt, loss={'regression':
'mse', 'class': 'binary_crossentropy'}, loss_weights=[1.,
0.2])
```

While training the network don't forget to add additional output as well:

```
history = final_model.fit(X_train, [Y_train, Y_train2],
              nb_epoch = 100,
              batch_size = 256,
              verbose=1,
              validation_data=(X_test, [Y_test, Y_test2]),
              callbacks=[reduce_lr, checkpointer],
              shuffle=True)
```

Here is general loss function graph (for both classification and regression loss in total):

Multitask network loss

And here is forecasting result.

Multitask network result

If we check main forecasting metrics, we have: the **MSE is 0.0161, MAE 0.070 is and MAPE is 2.85%**. We could do it! Really, adding an auxiliary loss helps to make better predictions! Don't forget to look on the full code.

> *Homework 1: try to observe if the same approach helps to make the performance of classification problem better.*

Mulimodal learning

Another illustration showing what multimodal learning is about

Let’s ask about this term in Wikipedia again:

> *The information in real world usually comes as different modalities. For example, images are usually associated with tags and text explanations; texts contain images to more clearly express the main idea of the article. Different modalities are characterized by very different statistical properties. For instance, images are usually represented as pixel intensities or outputs of feature extractors, while texts are represented as discrete word count vectors. Due to the distinct statistical properties of different information resources, it is very important to discover the relationship between different modalities.*

What we need to understand, that events in real world are happening due to different reasons, and the same holds to financial markets. You can be an expert of looking on the charts, but there are also other sources of information like news, gossips, insider information and actually all these things we have to take into account.

In our dataset we have text and time series. If we already know how to model dependencies in time series, text is a bit different. As we have some vector (OHLCV) for a time stamp on a chart, we want to have all available text in a form of a vector. There are a lot of different methods to do it: word2vec, doc2vec, Glove, bag of words models etc. We will use very straightforward and in general **incorrect** approach to transform our text to a vector. I just want to show how to build multimodal networks and that even this stupid approach works.

First, we will just concatenate all news headers for a single day that we have in our dataset. After, having all these merged headers we will train on all of them word2vec model, that learns to represent a single word as a vector of fixed dimension. And to represent a sentence, we simply will average all word2vec vectors of every word in it. **I want to underline, that in general it's incorrect way to work with text data and we do this just for simplicity and to show proof of a concept**:

```
def transform_text_into_vectors(train_text, test_text,
embedding_size = 100, model_path = 'word2vec10.model'):
 '''
  Transforms sentences into sequences of word2vec vectors
  Returns train, test set and trained word2vec model
 '''
 data_for_w2v = []
 for text in train_text + test_text:
     words = text.split(' ')
     data_for_w2v.append(words)


model = Word2Vec(data_for_w2v, size=embedding_size,
window=5, min_count=1, workers=4)
 model.save(model_path)
 model = Word2Vec.load(model_path)


train_text_vectors = [[model[x] for x in sentence.split('
')] for sentence in train_text]
 test_text_vectors = [[model[x] for x in sentence.split('
')] for sentence in test_text]


train_text_vectors = [np.mean(x, axis=0) for x in
train_text_vectors]
 test_text_vectors = [np.mean(x, axis=0) for x in
test_text_vectors]


return train_text_vectors, test_text_vectors, model
```

Full code of data processing is in the repo. And here is the code for a network with several inputs:

```
main_input = Input(shape=(30, 5), name='ts_input')
text_input = Input(shape=(30, 100), name='text_input')

lstm1 = LSTM(10, return_sequences=True,
recurrent_dropout=0.25, dropout=0.25,
bias_initializer='ones')(main_input)
lstm1 = LSTM(10, return_sequences=True,
recurrent_dropout=0.25, dropout=0.25,
bias_initializer='ones')(lstm1)
lstm1 = Flatten()(lstm1)

lstm2 = LSTM(10, return_sequences=True,
recurrent_dropout=0.25, dropout=0.25,
bias_initializer='ones')(text_input)
lstm2 = LSTM(10, return_sequences=True,
recurrent_dropout=0.25, dropout=0.25,
bias_initializer='ones')(lstm2)

lstm2 = Flatten()(lstm2)
lstms = average([lstm1, lstm2])

x = Dense(64)(lstms)
x = LeakyReLU()(x)
x1 = Dense(1, activation = 'linear', name='regression')(x)

final_model = Model(inputs=[main_input, text_input],
              outputs=[x1])
```

The most interesting point is merging representation learnt from text sequence and time series sequence. I used same dimensional representation on purpose, to show, that we can do several things with merged vectors: add them, average them or simply concatenating. We will obtain following results (I won't add plots, they looks the same anyway)

AVERAGE: **MSE is 0.0153, MAE 0.069 is and MAPE is 3.15%**

CONCATENATION: **MSE is 0.0158, MAE 0.07 is and MAPE is 3.01%**

ADDITION: **MSE is 0.0171, MAE 0.074 is and MAPE is 3.369%**

As we can see, results aren't better than our normal baseline. I think, it can be explained with the fact that our text representation is very silly, and, actually, it can even distract our model from useful time series data.

> *Homework 2: use doc2vec as a text feature extractor. How the performance changed?*

Multimodal + multitask learning

How a network with several heads and legs really looks like

This part will be really small: we will just check what will happen, if we will have two inputs (text + time series) and two outputs if our network. Will this monster do something useful?

```
main_input = Input(shape=(30, 5), name='ts_input')
text_input = Input(shape=(30, 100), name='text_input')
```

```
lstm1 = LSTM(10, return_sequences=True,
recurrent_dropout=0.25, dropout=0.25,
bias_initializer='ones')(main_input)
lstm1 = LSTM(10, return_sequences=True,
recurrent_dropout=0.25, dropout=0.25,
bias_initializer='ones')(lstm1)
lstm1 = Flatten()(lstm1)


lstm2 = LSTM(10, return_sequences=True,
recurrent_dropout=0.25, dropout=0.25,
bias_initializer='ones')(text_input)
lstm2 = LSTM(10, return_sequences=True,
recurrent_dropout=0.25, dropout=0.25,
bias_initializer='ones')(lstm2)
lstm2 = Flatten()(lstm2)


lstms = concatenate([lstm1, lstm2])


x1 = Dense(64)(lstms)
x1 = LeakyReLU()(x1)
x1 = Dense(1, activation = 'linear', name='regression')(x1)


x2 = Dense(64)(lstms)
x2 = LeakyReLU()(x2)
x2 = Dropout(0.9)(x2)
x2 = Dense(1, activation = 'sigmoid', name = 'class')(x2)


final_model = Model(inputs=[main_input, text_input],
              outputs=[x1, x2])
```

Here are results: **MSE is 0.0168, MAE 0.072 is and MAPE is 3.08%.** Seems like we should've stop at multitask experiments :)

Results plot and loss plot for multimodal-multitask network

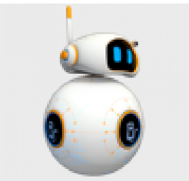
> *Homework 3: having several tasks and inputs needs more training time—check the performance after 500 epochs.*

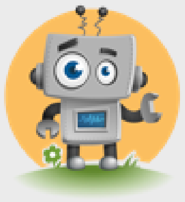
Conclusions

As for me, it was the most interesting experiment out of the whole series. It wasn't just about straightforward training a network, but we really used smart approach to support one main task (volatility forecasting) with another one (classification) and tasted one of the most promising areas of modern machine learning—multimodal learning.

From practical point of view we can see, that using several losses is good and correctly working idea, but while learning with different sources we really need to work a lot on all data—we didn't care much about our text as we did with time series, and that's why we couldn't achieve better results. I encourage readers to try other techniques to represent text data and check the performance—I am sure you will be surprised ;)

Stay tuned, there are lot of other amazing topics to check out!


Join the Community


Subscribe

Apply To Be A Writer