

Factor models via Autoencoders

A clever way of using Neural Networks to solve a familiar but important problem in Finance was proposed by [Gu, Kelly, and Xiu, 2019](https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3335536)
(https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3335536).

It is an extension of the Factor Model framework of Finance, combined with the tools of dimensionality reduction (to find the factors) of Deep Learning: the Autoencoder.

Factor Model review

We will begin with a quick review/introduction to Factor Models in Finance.

The universe of securities (e.g., equities) is often quite large

- several hundred (or thousands) of individual tickers
- denote the size by n

It is often the case that the returns of many securities can be explained

- as being the sum of influences of "common factors"
 - market index
 - industry indices
 - size, momentum

It is sometimes useful to *approximate* the return of a security

- as the dot product of
- the sensitivity of the security to a number f of *common factors*
- the returns of the common factors

This is useful

- as a means of *dimensionality* reduction
 - we need timeseries of returns for only $f \leq n$ factors rather than all n securities
- as a means of understanding the behavior of two or more securities
 - as the sum of common influences
 - rather than completely idiosyncratic returns
 - Hedging, risk-management

First, some necessary notation:

- $\mathbf{r}_s^{(d)}$: Return of ticker s on day d .
- $\hat{\mathbf{r}}_s^{(d)}$: approximation of $\mathbf{r}_s^{(d)}$
- n_{tickers} : **large** number of tickers
- n_{dates} : number of dates
- n_{factors} : **small** number of factors: independent variables (features) in our approximation
- Matrix \mathbf{R} of ticker returns, indexed by *date*
 - $\mathbf{R} : (n_{\text{dates}} \times n_{\text{tickers}})$
 - $||\mathbf{R}^{(d)}|| = n_{\text{tickers}}$
 - $\mathbf{R}^{(d)}$ is vector of returns for each of the n_{tickers} on date d
- \mathbf{r} will denote a vector of single day returns: $\mathbf{R}^{(d)}$ for some date d

Notation summary

| term | meaning | | |
|--|--|---------------------|---|
| s | ticker | | |
| n_{tickers} | number of tickers | | |
| d | date | | |
| n_{dates} | number of dates | | |
| n_{chars} | number of characteristics per ticker | | |
| m | number of examples | | |
| | $m = n_{\text{dates}}$ | | |
| i | index of example | | |
| | There will be one example per date, so we use i and d interchangeably. | | |
| $[\mathbf{X}^{(i)}, \mathbf{R}^{(i)}]$ | example i | | |
| | $\$$ | \mathbb{X}^{ip} | $= (n_{\text{tickers}} \times n_{\text{chars}}) \$$ |
| | $\$$ | \mathbb{R}^{ip} | $= n_{\text{tickers}} \$$ |
| $\mathbf{X}_s^{(d)}$ | vector of ticker s 's characteristics on day d | | |
| | $\$$ | \mathbb{X}^{dp_s} | $= n_{\text{chars}} \$$ |

Note

The paper actually seeks to predict $\hat{\mathbf{r}}_s^{(d+1)}$ (forward return) rather than approximate the current return $\hat{\mathbf{r}}_s^{(d)}$.

We will present this as an approximation problem as opposed to a prediction problem for simplicity of presentation (i.e., to include PCA as a model).

A **factor model** seeks to approximate/explain the return of a *number* of tickers in terms of common "factors" \mathbf{F}

- $\mathbf{F} : (n_{\text{dates}} \times n_{\text{factors}})$
$$\mathbf{R}_1^{(d)} = \beta_1^{(d)} \cdot \mathbf{F}^{(d)} + \epsilon_1$$
$$\vdots$$
$$\mathbf{R}_{n_{\text{tickers}}}^{(d)} = \beta_{n_{\text{tickers}}}^{(d)} \cdot \mathbf{F}^{(d)} + \epsilon_{n_{\text{tickers}}}$$

There are several ways to create a factor model

- depending on what we assume
- is given, in addition to \mathbf{R}

We will examine each method, but here is a high-level summary:

| Name | Given | Solve for |
|---------------------------|---|--|
| Pre-defined factors | $\mathbf{F} : (n_{\text{dates}} \times n_{\text{factors}})$ | $\beta_s : (n_{\text{factors}})$ |
| Pre-defined sensitivities | $\beta : (n_{\text{tickers}} \times n_{\text{factors}})$ | $\mathbf{F}^{(d)}$ |
| Nothing pre-defined | | \mathbf{F}, β^T |
| AE for cond. risk factors | | $\mathbf{F}^{(d)}, \underline{\beta}^{(d)} : (n_{\text{tickers}} \times n_{\text{factors}})$ |
| | | time-varying $\underline{\beta} : (n_{\text{tickers}} \times n_{\text{factors}})$ |

The first two approaches

- take one part (e.g., sensitivities or factor returns) of the product as given
- solves for the other part

The PCA approach

- solves for *both* parts of the product
 - subject to the ticker sensitivities β being fixed through time

The Autoencoder for Conditional Risk factors approach

- solves for *both* parts of the product
 - **and** has time-varying sensitivities β and factor returns \mathbf{F}

Pre-defined factors, solve for sensitivities

Suppose \mathbf{F} is given: a matrix of returns of "factors" over a range of dates

- $\mathbf{F}^{(d)}$ includes the returns of multiple factor tickers
 - e.g., market, several industries, large/small cap indices

Solve for β_s , for each s

- n_{tickers} separate Linear Regression models
- Linear regression for ticker s :
 - r_s and \mathbf{F} are time series (length n_{dates}) of returns for tickers/factors
 - Solve for β_s
 - constant over time

$$\mathbf{r}_s = \begin{pmatrix} \mathbf{r}_s^{(1)} \\ \mathbf{r}_s^{(2)} \\ \vdots \\ \mathbf{r}_s^{(n_{\text{dates}})} \end{pmatrix}, \mathbf{F} = \begin{pmatrix} \mathbf{F}_1^{(1)} & \dots & \mathbf{F}_{n_{\text{factors}}}^{(1)} \\ \mathbf{F}_1^{(2)} & \dots & \mathbf{F}_{n_{\text{factors}}}^{(2)} \\ \vdots & & \vdots \\ \mathbf{F}_1^{(n_{\text{dates}})} & \dots & \mathbf{F}_{n_{\text{factors}}}^{(n_{\text{dates}})} \end{pmatrix}, \beta_s = \begin{pmatrix} \beta_{s,1} \\ \beta_{s,2} \\ \vdots \\ \beta_{s,n_{\text{factors}}} \end{pmatrix}$$

$$\mathbf{r}_s = \mathbf{F} * \beta_s$$

Pre-defined sensitivities, solve for factors

Suppose β is given:

- for each ticker s : $\beta_{s,j}$ is the sensitivity of s to \mathbf{F}_j

Solve for $\mathbf{F}^{(d)}$ for each d

- n_{dates} separate Linear Regressions
- Linear regression for date d
 - $\mathbf{r}^{(d)}$ and $\beta^{(d)}$ are cross sections (width n_{tickers}) of one day ticker returns/sensitivities
 - Solve for $\mathbf{F}^{(d)}$
 - constant over tickers

$$\mathbf{r}^{(d)} = \begin{pmatrix} \mathbf{r}_1^{(d)} \\ \mathbf{r}_1^{(d)} \\ \vdots \\ \mathbf{r}_{n_{\text{tickers}}}^{(d)} \end{pmatrix}, \quad \mathbf{F}^{(d)} = \begin{pmatrix} \mathbf{F}_1^{(d)} \\ \mathbf{F}_2^{(d)} \\ \vdots \\ \mathbf{F}_{n_{\text{factors}}}^{(d)} \end{pmatrix}, \quad \beta = \begin{pmatrix} \beta_{1,1}, & \dots & \beta_{1,n_{\text{factors}}} \\ \beta_{2,1}, & \dots & \beta_{2,n_{\text{factors}}} \\ \vdots & & \\ \beta_{n_{\text{tickers}},1}, & \dots & \beta_{n_{\text{tickers}},n_{\text{factors}}} \end{pmatrix}$$

$$\mathbf{F}_s^{(d)} = \mathbf{F}^{(d)}$$

$$\langle \mathbf{X}^{(s)}, \mathbf{y}^{(s)} \rangle = \langle \beta^{(s)}, \mathbf{r}^{(s)} \rangle$$

$$\mathbf{r}^{(d)} = \beta * \mathbf{F}^{(d)}$$

Solve for sensitivities and factors: PCA

Yet another possibility: solve for β and \mathbf{F} *simultaneously*.

Recall Principal Components

- Representing \mathbf{X} (defined relative to n_{tickers} "standard" basis vectors) via an *alternate basis* \mathbf{V}

$$\mathbf{X} = \tilde{\mathbf{X}} \mathbf{V}^T$$

In our case: we will factor \mathbf{R} .

In this case, we identify \mathbf{X} with the returns \mathbf{R} . Thus, without dimensionality reduction:

$$\mathbf{R} = \tilde{\mathbf{R}}\mathbf{V}^T$$

where

$$\mathbf{R}, \tilde{\mathbf{R}} : (n_{\text{dates}} \times n_{\text{tickers}})$$

$$\mathbf{V}^T : (n_{\text{tickers}} \times n_{\text{tickers}})$$

We will choose the second factorization.

- create a timeseries of returns for factors
- the sensitivity of a ticker to a factor remains constant over time

$$\mathbf{R} \approx \mathbf{F} \beta^T$$

- $\mathbf{F}^T : (n_{\text{dates}} \times n_{\text{factors}})$
 - the "alternative" basis: the "factors"
 - is \mathbf{V} with columns eliminated b/c of dimensionality reduction
- $\beta^T : (n_{\text{factors}} \times n_{\text{tickers}})$
 - so $\beta^{(s)}$ are sensitivities of s to factors
- Solve for \mathbf{F}, β simultaneously

\mathbf{r}_s , the time series of returns of ticker s is approximated by a combination of returns of n_{factors} .

$$\mathbf{r}_s^{(d)} = \beta_s * \mathbf{F}^{(d)}$$

- $\beta_s^{(d)}$ is constant over time
 $\beta_s^{(d)} = \beta_s$

The daily observation of n_{tickers} returns $\mathbf{R}^{(d)}$ is replaced by n_{factors} returns $\mathbf{F}^{(d)}$

PCA seeks to *approximate* \mathbf{R} with fewer than n_{tickers} basis vectors

- this is the *dimensionality reduction*
- reduce from n_{tickers} dimensions to n_{factors} dimensions

This paper

This paper will create a factor model that

- Solve for \mathbf{F} , β simultaneously
 - like PCA
 - but with time-varying β

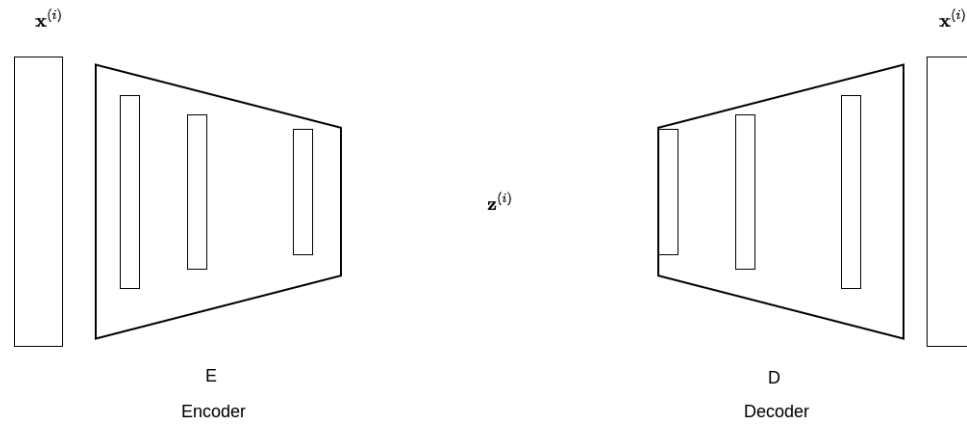
This very general approach is facilitated because

- \mathbf{F} and β are defined by Neural Networks

Autoencoder

The paper refers to the model as a kind of Autoencoder.

Autoencoder



Let's review the topic.

- An Autoencoder has two parts: an Encoder and a Decoder
- The Encoder maps inputs $\mathbf{x}^{(i)}$, of length n
- Into a "latent vectors" $\mathbf{z}^{(i)}$ of length $n' \leq n$
- If $n' < n$, the latent vector is a *bottleneck*
 - reduced dimension representation of $\mathbf{x}^{(i)}$
- The Decoder maps $\mathbf{z}^{(i)}$ into $\hat{\mathbf{x}}^{(i)}$, of length n , that is an approximation of $\mathbf{x}^{(i)}$

The training examples for an Autoencoder are

$$\langle \mathbf{X}^{(d)}, \mathbf{y}^{(d)} \rangle = \langle \mathbf{R}^{(d)}, \mathbf{R}^{(d)} \rangle$$

That is

- we want the output for each example to be identical to the input

The challenge:

- the input is passed through a "bottleneck" \mathbf{z} of lower dimensions than the example length n
- information is lost
- analog: using PCA for dimensionality reduction, but with non-linear operations

Autoencoder for Conditional Risk Factors

Imagine that we are given $\mathbf{R} : (n_{\text{dates}} \times n_{\text{tickers}})$

- timeseries (length n_{dates}) of returns of n_{tickers} tickers

Suppose we map a one day set of returns $\mathbf{R}^{(d)}$ into two separate values

- $\beta^{(d)} : (n_{\text{tickers}} \times n_{\text{factors}})$ -- the sensitivity of each ticker to each of n_{factors} one day "factor" returns
- $\mathbf{F}^{(d)} : (n_{\text{factors}} \times 1)$ -- the one day returns of n_{factors} factors

Our goal is to output $\hat{\mathbf{R}}^{(d)}$, an approximations of $\mathbf{R}^{(d)}$ such that

$$\hat{\mathbf{R}}^{(d)} = \beta^{(d)} * \mathbf{F}^{(d)}$$

$$\hat{\mathbf{R}}^{(d)} \approx \mathbf{R}^{(d)}$$

This is the same goal as an Autoencoder but subject to the constraint that $\hat{\mathbf{R}}^{(d)}$

- is the product of the ticker sensitivities and factor returns

The Neural Network *simultaneously* solves for $\beta^{(d)}$ and $\mathbf{F}^{(d)}$.

This looks somewhat like PCA

- **but**, in PCA, β does not vary by day: it is constant over days
- in this model, $\beta^{(d)}$ varies by day

This paper goes one step further than the standard Autoencoder

- Inputs \mathbf{X}
: $(n_{\text{dates}}$
 $\times n_{\text{tickers}}$
 $\times n_{\text{chars}})$
- rather than \mathbf{R}
: $(n_{\text{dates}}$
 $\times n_{\text{tickers}})$

Each ticker s on each day d , has $n_{\text{chars}} \geq 1$ "characteristic"

- one of them may be the daily return $\mathbf{R}^{(d)}$
- but may also include a number of other time varying characteristics

The proposed model is a Neural Network with two sub-networks.

The *Beta network* computes $\beta_s^{(d)} = \text{NN}_\beta(\mathbf{X}_s^{(d)}; \mathbf{W}_\beta)$

- $\mathbf{X}_s^{(d)}$ as input
- parameterized by weights \mathbf{W}_β
- $\beta_s^{(d)}$ is only a function of $\mathbf{X}_s^{(d)}$, the characteristics of s
 - and **not** of any other ticker $s' \neq s$
 - $\beta_s^{(d)}$ shares \mathbf{W}_β across **all** tickers s' and dates d'
 - contrast this with factor model with fixed factors
 - we solve for a separate β_s for each ticker s
 - via per-ticker timeseries regression
 - contrast this with PCA
 - β_s is influenced by $\mathbf{R}_{s'}$ for $s' \neq s$

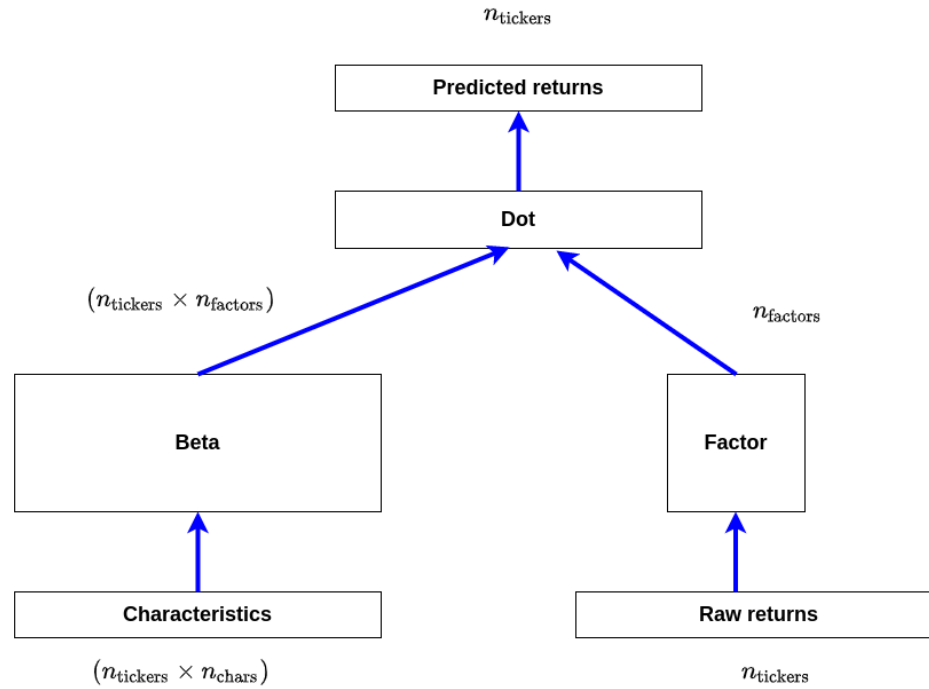
The *Factor network* computes $\mathbf{F}^{(d)} = \text{NN}_{\mathbf{F}}(\mathbf{R}^{(d)}, \mathbf{W}_{\mathbf{F}})$

- $\mathbf{R}^{(d)}$ as input (not $\mathbf{X}^{(d)}$ as in the Beta network)
- parameterized by weights $\mathbf{W}_{\mathbf{F}}$ $\mathbf{R}^{(d)}$ is only a function of $\mathbf{R}^{(d)}$ for date d
 - and **not** of any other date $d' \neq d$
 - $\mathbf{F}^{(d)}$ shares $\mathbf{W}_{\mathbf{F}}$ across **all** dates

This model

- has *neither* pre-defined Factors \mathbf{F} or pre-defined Sensitivities β
- Simultaneously solve for $\beta_s^{(d)}$ and $\mathbf{F}^{(d)}$

Here is a picture



Summary of this paper

Approximate cross section of daily returns: $\hat{\mathbf{r}}^{(d)} \approx \mathbf{r}^{(d)}$
 $\mathbf{r}^{(d)} \approx \hat{\mathbf{r}}^{(d)} = \boldsymbol{\beta}^{(d)} * \mathbf{F}^{(d)}$

- like an Autoencoder
- subject
 - to returns as product of sensitivities and factors: $\hat{\mathbf{r}}^{(d)} = \boldsymbol{\beta}^{(d)} * \mathbf{F}^{(d)}$
 - $\boldsymbol{\beta}_s^{(d)} = \text{NN}_{\boldsymbol{\beta}}(\mathbf{X}_s^{(d)}; \mathbf{W}_{\boldsymbol{\beta}})$
 - $\mathbf{F}^{(d)} = \text{NN}_{\mathbf{F}}(\mathbf{R}^{(d)}, \mathbf{W}_{\mathbf{F}})$

Shapes:

- $\mathbf{r}^{(d)} : (n_{\text{tickers}} \times 1)$
- $\boldsymbol{\beta} : (n_{\text{tickers}} \times n_{\text{factors}})$
- $\mathbf{F}^{(d)} : (n_{\text{factors}} \times 1)$

Complete Neural Network

Beta (Input) side of network

The Beta network NN_β

- maps *ticker characteristics* to *ticker factor sensitivities*
 - for each day

It uses a single layer fully connected (Dense) Layer with n_{factors} units

- input: n_{chars} attributes (characteristics) for each of n_{tickers} tickers
- output: n_{factors} factor sensitivities for each of n_{tickers} tickers

$$\text{NN}_{\beta} : (n_{\text{tickers}} \times n_{\text{chars}}) \mapsto (n_{\text{tickers}} \times n_{\text{factors}})$$

Input \mathbf{X}

$$\mathbf{X} : (n_{\text{dates}} \times n_{\text{tickers}} \times n_{\text{chars}})$$

$$\mathbf{X}^{(d)} : (n_{\text{tickers}} \times n_{\text{chars}})$$

- Example on date d
- Consists of n_{tickers} tickers, each with n_{chars} characteristics

Sub Neural network NN_β

$$\text{NN}_\beta = \text{Dense}(n_{\text{factors}})(\mathbf{X})$$

- Fully connected network
- $\text{Dense}(n_{\text{factors}})$ computes a function $(n_{\text{tickers}} \times n_{\text{chars}}) \mapsto (n_{\text{tickers}} \times n_{\text{factors}})$
- Threads over ticker dimension ([see](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense)
https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense)
 - tickers share same weights across **all** tickers
 - single $\text{Dense}(n_{\text{factors}})$ **not** n_{tickers} copies of $\text{Dense}(n_{\text{factors}})$ with independent weights

$$\mathbf{W}_\beta : (n_{\text{factors}} \times n_{\text{chars}})$$

- weights shared across all d, s
 - $\mathbf{W}_{\beta,s}^{(d)} = \mathbf{W}_{\beta,s'}^{(d')}$ for all s', d'
 - the transformation of characteristics to beta *independent* of ticker
- hence, size of \mathbf{W}_β is $(n_{\text{factors}} \times n_{\text{chars}})$

$$\beta^{(d)} = \text{Dense}(n_{\text{factors}})(\mathbf{X}^{(d)})$$

$$\beta^{(d)} : (n_{\text{tickers}} \times n_{\text{factors}})$$

Factor side of network

The Factor network NN_F

- maps *ticker returns* to *factor returns*
 - **for each day**

It uses a single layer fully connected (Dense) Layer with n_{factors} units

- input: vector of ticker returns (one-day)
- output: vector of factor returns

$$\text{NN}_{\mathbf{F}} : n_{\text{tickers}} \mapsto n_{\text{factors}}$$

Input \mathbf{R}

$$\mathbf{R} : (n_{\text{dates}} \times n_{\text{tickers}})$$

$$\mathbf{R}^{(d)} : (n_{\text{tickers}} \times 1)$$

- Example on date d
- Consists of returns of n_{tickers} tickers

Sub Neural network $\mathbf{NN_F}$

$$\mathbf{NN_F} = \text{Dense}(n_{\text{factors}})$$

- Fully connected network
- $\text{Dense}(n_{\text{factors}})$ computes a function $n_{\text{tickers}} \mapsto n_{\text{factors}}$

$$\mathbf{W}_{\mathbf{F}} : (n_{\text{factors}} \times n_{\text{tickers}})$$

- Weights shared across all d, s
 - $\mathbf{W}_{\mathbf{F},s}^{(d)} = \mathbf{W}_{\mathbf{F},s'}^{(d')}$ for all s', d'
 - the transformation of cross section of ticker returns to Factor returns *independent* of ticker
- hence, size of $\mathbf{W}_{\mathbf{F}}$ is $(n_{\text{tickers}} \times n_{\text{factors}})$

$$\mathbf{F}^{(d)} = \text{Dense}(n_{\text{factors}})(\mathbf{R}^{(d)})$$

$$\mathbf{F}^{(d)} : n_{\text{factors}}$$

Dot

The Dot layer computes the dot product of tickers sensitivities and factor returns.

- this is the predicted return

$$\hat{\mathbf{r}}^{(d)} = \boldsymbol{\beta}^{(d)} \cdot \mathbf{F}^{(d)}$$

Dot product threads over factor dimension

- Computes $\hat{\mathbf{r}}_s^{(d)} = \beta_s^{(d)} \cdot \mathbf{F}^{(d)}$ for each s
 - each s is a row of $\boldsymbol{\beta}^{(d)}$

$$\hat{\mathbf{r}}^{(d)} : n_{\text{tickers}}$$

Loss

The key to any NN is the Loss Function.

Let $\mathcal{L}_{(s)}^{(d)}$ denote error of ticker s on day d .

$$\mathcal{L}_{(s)}^{(d)} = \mathbf{r}_s^{(d)} - \hat{\mathbf{r}}_s^{(d)}$$

$\mathcal{L}^{(d)}$ is the loss, across tickers, on date d (one training example)

$$\mathcal{L}^{(d)} = \sum_s \mathcal{L}_{(s)}^{(d)}$$

The number of examples m equals n_{dates}

So the Total Loss is

$$\mathcal{L} = \sum_d \mathcal{L}^{(d)}$$

Predicting future returns, rather than explaining contemporaneous returns

The model is sometimes presented as predicting **day ahead** returns rather than contemporaneous returns.

In that case the objective is

$$\hat{\mathbf{r}}^{(d)} = \mathbf{r}^{(d+1)}$$

and Loss for a single ticker and date becomes

$$\mathcal{L}_{(s)}^{(d)} = \mathbf{r}_s^{(d+1)} - \hat{\mathbf{r}}_s^{(d)}$$

Code

The model is built by the function `make_model`
([06 conditional autoencoder for asset pricing model.ipynb#Automate-model-generation](#)).

```
def make_model(hidden_units=8, n_factors=3):
    input_beta = Input((n_tickers, n_characteristics), name='input_beta')
    input_factor = Input((n_tickers,), name='input_factor')

    hidden_layer = Dense(units=hidden_units, activation='relu', name='hidden_layer')(input_beta)
    batch_norm = BatchNormalization(name='batch_norm')(hidden_layer)

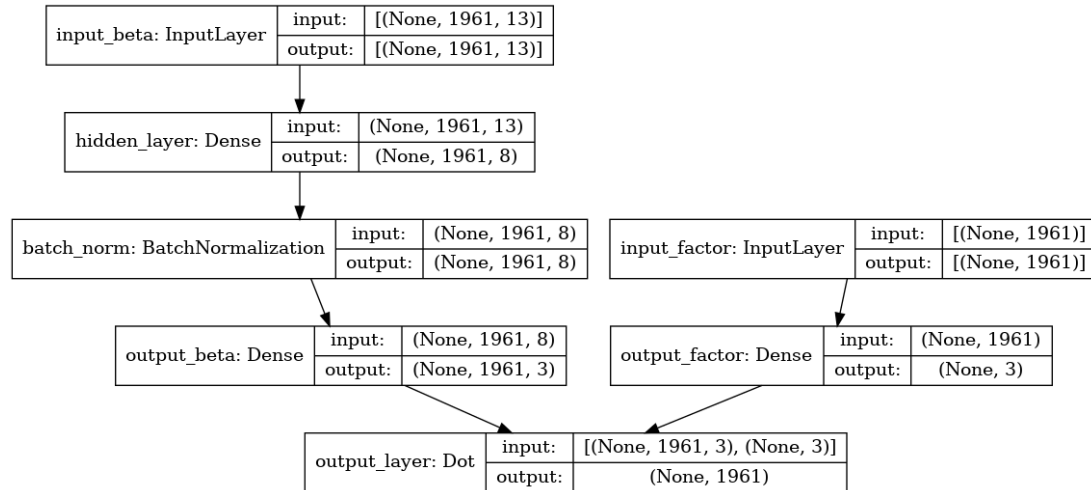
    output_beta = Dense(units=n_factors, name='output_beta')(batch_norm)

    output_factor = Dense(units=n_factors, name='output_factor')(input_factor)

    output = Dot(axes=(2,1), name='output_layer')([output_beta, output_factor])

    model = Model(inputs=[input_beta, input_factor], outputs=output)
    model.compile(loss='mse', optimizer='adam')
    return model
```

Here is what the model looks like:



Highlights

- **Two** input layers
 - one each for the Beta and Factor networks
- The model is passed a **pair** as input
 - one input for each side of the network

```
Model(inputs=[input_beta, input_factor], outputs=output)
```

- and is called
[\(06_conditional_autoencoder_for_asset_pricing_model.ipynb#Trai Model\)](#) with a pair

```
model.fit([X1_train, X2_train], y_train,  
        ...
```

- Loss function: MSE

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

Training data

```
def get_train_valid_data(data, train_idx, val_idx):
    train, val = data.iloc[train_idx], data.iloc[val_idx]
    X1_train = train.loc[:, characteristics].values.reshape(-1, n_tickers, n_characteristics)
    X1_val = val.loc[:, characteristics].values.reshape(-1, n_tickers, n_characteristics)
    X2_train = train.loc[:, 'returns'].unstack('ticker')
    X2_val = val.loc[:, 'returns'].unstack('ticker')
    y_train = train.returns_fwd.unstack('ticker')
    y_val = val.returns_fwd.unstack('ticker')
    return X1_train, X2_train, y_train, X1_val, X2_val, y_val
```

- `X1_train`: ticker characteristics

```
X1_train = train.loc[:, characteristics].values.reshape(-1, n_tickers,
n_characteristics)
```

- `X2_train`: ticker returns

- Dataframe attribute `returns`

```
X2_train = train.loc[:, 'returns'].unstack('ticker')
```

- `y_train`: *forward* ticker returns

- Dataframe attribute `returns_fwd`

```
y_train = train.returns_fwd.unstack('ticker')
```


In [2]: `print("Done")`

Done

