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.

You can find code (https://github.com/stefan-jansen/machine-learning-for-trading/blob/main/20_autoencoders_for_conditional_risk_factors/06_conditional_autoenc for this model as part of the excellent book by Stefan Jansen (https://github.com/stefan-jansen/machine-learning-for-trading/blob/main/20_autoencoders_for_conditional_risk_factors/06_conditional_autoenc

- Github (https://github.com/stefan-jansen/machine-learning-for-trading)
- In order to run the code notebook, you first need to run a notebook for data prepara jansen/machine-learning-for-trading/blob/main/20_autoencoders_for_conditional_risk_factors/05_conditional_a
  - This notebook relies on files created by notebooks from earlier chapters of
  - So, if you want to run the code, you have a lot of preparatory work ahead of
  - Try to take away the ideas and the coding

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.		
$[\backslash\mathbf{X}^{\backslash\mathrm{ip}}, \mathbf{R}^{\backslash\mathrm{ip}}]$	example i		
	$	\X^\ip	= (\ntickers \times \nchars)$
	$	\R^\ip	= \ntickers$
$\backslash\mathbf{X}_s^{(d)}$	vector of ticker s's characteristics on day d		
	$	\X^\dp_s	= \nchars$

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" $\mathbf{F}$

- $\mathbf{F} : (n_{\text{dates}} \times n_{\text{factors}})$

$$
\begin{aligned}
\mathbf{R}_1^{(d)} &= \beta_1^{(d)} \cdot \mathbf{F}^{(d)} + \epsilon_1 \\
\vdots \\
\mathbf{R}_{n_{\text{tickers}}}^{(d)} &= \beta_{n_{\text{tickers}}}^{(d)} \cdot \mathbf{F}^{(d)} + \epsilon_{n_{\text{tickers}}}
\end{aligned}
$$

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		$\mathbf{F}, \beta^T$
AE for cond. risk factors		$\mathbf{F}^{(d)}, \underline{\beta}^{(d)} : (n_{\text{tickers}} \times n_{\text{factors}})$ time-varying $\underline{\beta} : (n_{\text{tickers}} \times n$

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

$$\beta_s^{(d)} = \beta_s$$

- $\langle \backslash \mathbf{X}^{(d)}, \backslash \mathbf{y}^{(d)} \rangle = \langle \mathbf{F}^{(d)}, \mathbf{r}_s^{(d)} \rangle$

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

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

Picture of linear regression

- One ticker s at a time, as a timeseries
    - selected column in left matrix
- *Given*
    - matrix $\mathbf{F}$ of factor timeseries
        - columns of right matrix
- *Solve*
    - for sensitivities β_s (middle vector)
    - dot product of sensitivity vector and *row* of factors on *one date(
        - estimated returns

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

Linear regression solves for β_s

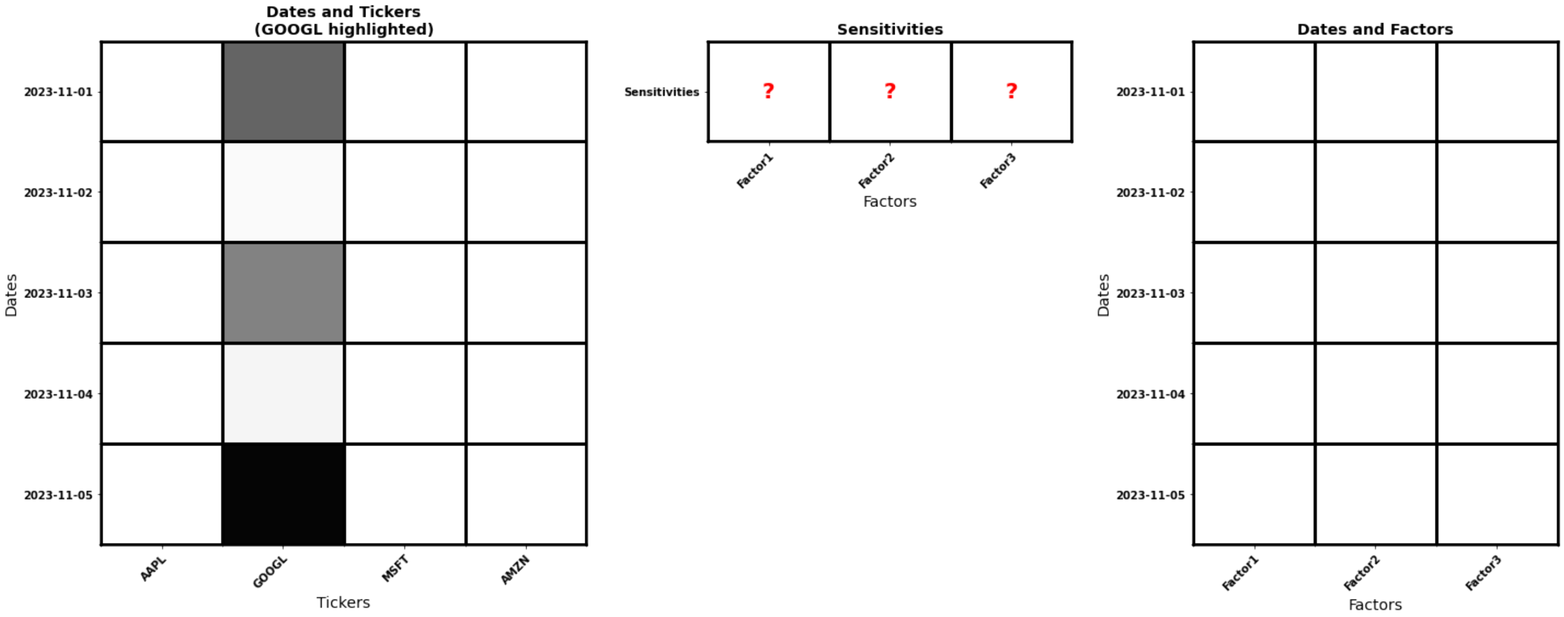
- to minimize errors *across dates*

$$\sum_{d=1}^{n_{\text{dates}}} \left(\mathbf{r}_s^{(d)} - \hat{\mathbf{r}}_s^{(d)} \right)^2$$

In [3]:

```
fig
```

Out[3]:



Dates and Tickers
(GOOGL highlighted)
2023-11-01
2023-11-02
2023-11-03
2023-11-04
2023-11-05
Dates
AAPL
GOOGL
MSFT
AMZN
Tickers
Sensitivities
Sensitivities
?
?
?
Factor1
Factor2
Factor3
Factors
Dates and Factors
2023-11-01
2023-11-02
2023-11-03
2023-11-04
2023-11-05
Dates
Factor1
Factor2
Factor3
Factors

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

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

  - $\langle \color{red}{\backslash\mathrm{X}}^{(s)}, \color{red}{\backslash\mathrm{y}}^{(s)} \rangle = \langle \beta^{(s)}, \mathbf{r}^{(s)} \rangle$

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

Picture of Cross-Sectional regression

- One date d at a time
    - selected row of left matrix

- *Given*

    - matrix β of sensitivities of each ticker to each factor
$$\beta : (n_{\text{tickers}} \times n_{\text{factors}})$$

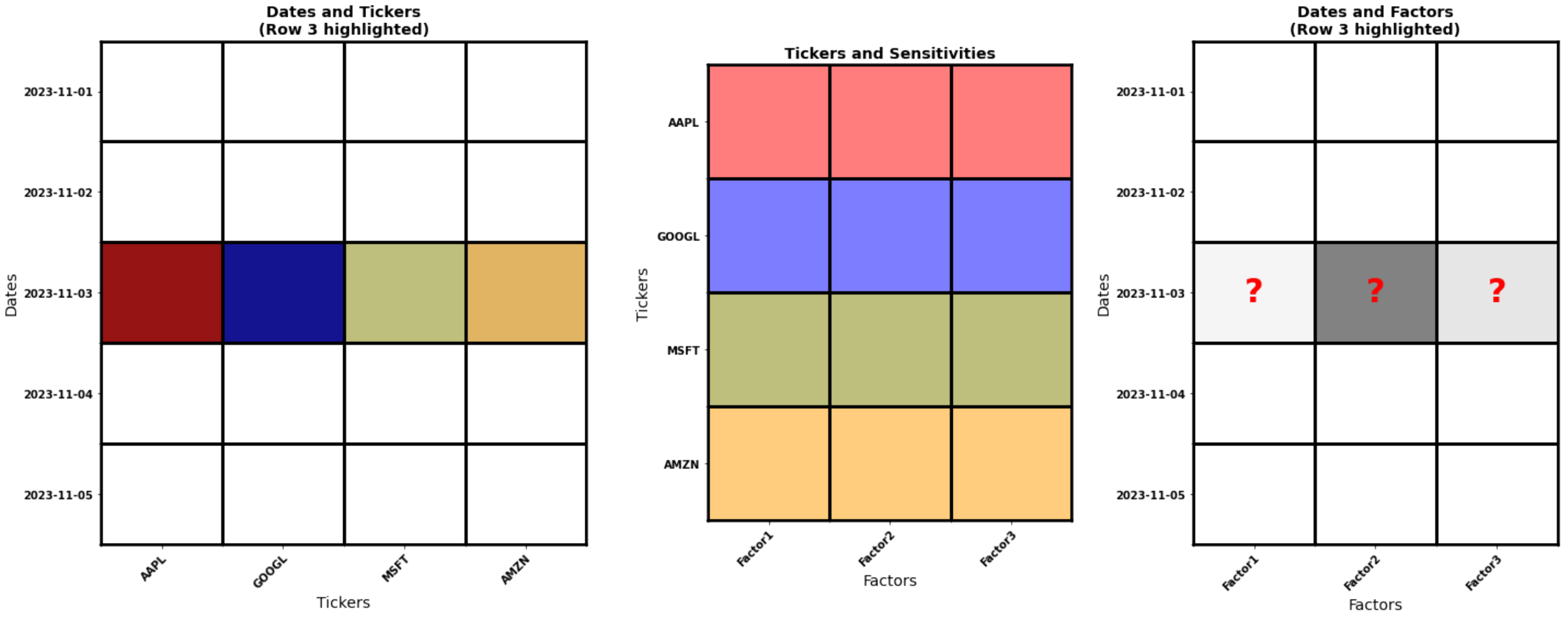
    - sensitivities are *constant* through time

- *Solve*
    - for factor returns at one date
        - selected row of right matrix
            - dot product of
            - sensitivity for one ticker (e.g., red row for AAPL)
            - factor returns at date d
            - give estimated return of AAPL
$$\hat{\mathbf{r}}_{\text{AAPL}} = \beta^{\text{AAPL}} \cdot \mathbf{F}^{(d)}$$

Cross-sectional regression solves for $\mathbf{F}^{(d)}$

In [5]:
```
fig
```

Out[5]:



Dates and Tickers
(Row 3 highlighted)
2023-11-01
2023-11-02
2023-11-03
2023-11-04
2023-11-05
Dates
AAPL
GOOGL
MSFT
AMZN
Tickers
Tickers and Sensitivities
AAPL
GOOGL
MSFT
AMZN
Tickers
Factor1
Factor2
Factor3
Factors
Dates and Factors
(Row 3 highlighted)
2023-11-01
2023-11-02
2023-11-03
2023-11-04
2023-11-05
Dates
?
?
?
Factor1
Factor2
Factor3
Factors

Solve for sensitivities and factors: PCA

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

Recall Principal Components

- Representing $\backslash\mathrm{X}$ (defined relative to n_{tickers} "standard" basis vectors) via an *alternate basis* $\mathbf{V}$
$$\backslash\mathrm{X} = \backslash\tilde{\mathrm{X}}\mathbf{V}^T$$

We factor matrix $\backslash\mathrm{X}$ into $\backslash\tilde{\mathrm{X}}$ and $\mathbf{V}^T$.

In our case, we identify

- $\backslash\mathrm{X}$ with the ticker returns $\mathbf{R}$.
- $\backslash\tilde{\mathrm{X}}$ with the factor returns $\mathbf{F}$
- $\mathbf{V}^T$ as β

Thus

$$\mathbf{R} = \tilde{\mathbf{R}}\beta$$

where

$$\begin{aligned}
&\mathbf{R}, \tilde{\mathbf{R}} : (n_{\text{dates}} \times n_{\text{tickers}}) \\
&\beta : (n_{\text{tickers}} \times n_{\text{tickers}})
\end{aligned}$$

The factorization is often used to achieve *dimensionality reduction*

- approximating the n_{tickers} timeseries
- with $n_{\text{factors}} < n_{\text{tickers}}$ factors

Reducing the dimension yields an approximation of $\mathbf{R}$

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

where

$$\begin{aligned}
\mathbf{R} &: (n_{\text{dates}} \times n_{\text{tickers}}) \\
\mathbf{F} &: (n_{\text{dates}} \times n_{\text{factors}}) \\
\beta &: (n_{\text{factors}} \times n_{\text{tickers}})
\end{aligned}$$

Thus

- column j of $\mathbf{F}$ is the return series of the j^{th} factor
- column j of β are the sensitivities of ticker i to the factors
    - which *don't* vary with time

The return $\mathbf{R}_j^{(d)}$ of ticker j on date d is approximated by

- the dot product of row d of F
    - the returns of the n_{factors} on date d
- and column j of β
    - the sensitivities of j to the n_{factors}

This paper

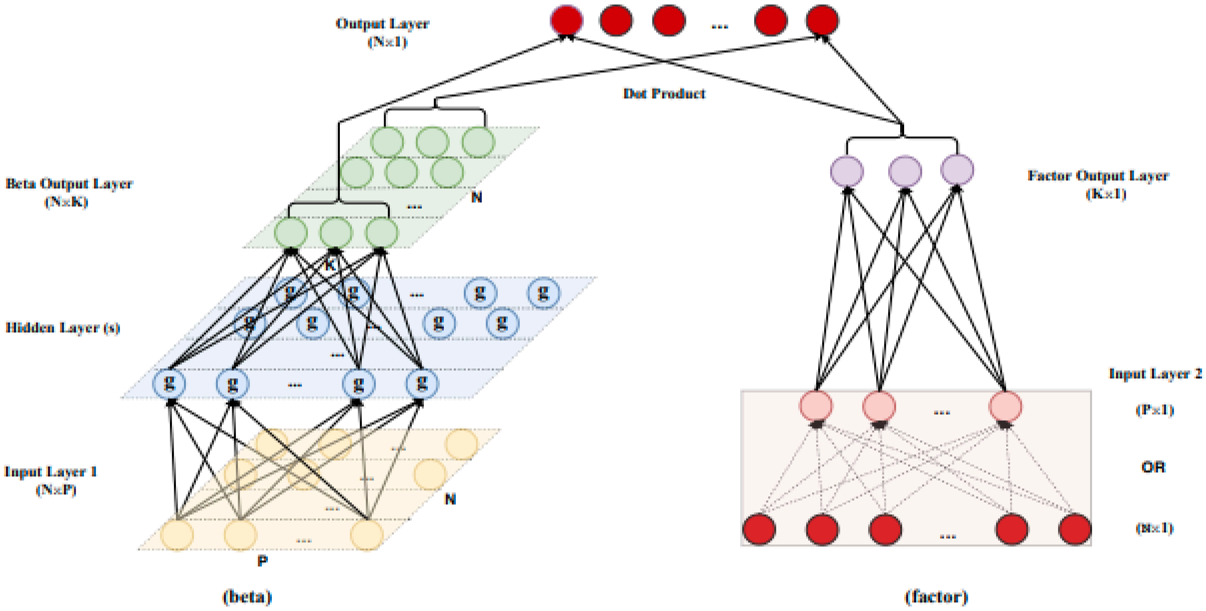
This paper will create a factor model that

- 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

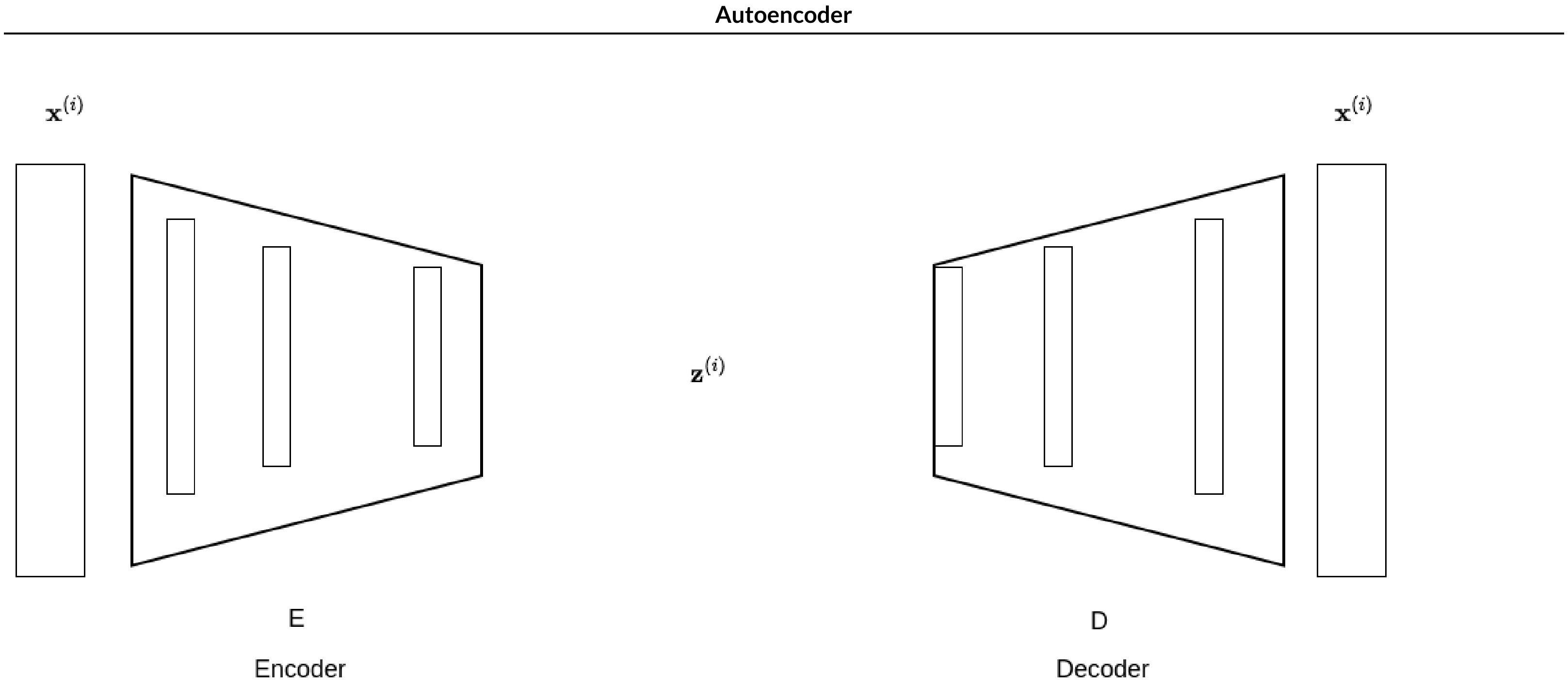
Figure 2: Conditional Autoencoder Model



Note: This figure presents the diagram of an autoencoder augmented to incorporate covariates in the factor loading specification. The left-hand side describes how factor loadings β_{t-1} at time $t-1$ (in green) depend on firm characteristics Z_{t-1} (in yellow) of the input layer 1 through an activation function g on neurons of the hidden layer. Each row of yellow neurons represents the $P \times 1$ vector of characteristics of one ticker. The right-hand side describes the corresponding factors at time t. f_t nodes (in purple) are weighted combinations of neurons of the input layer 2, which can either be P characteristic-managed portfolios x_t (in pink) or N individual asset returns r_t (in red). In the latter case, the input layer 2 is exactly what the output layer aims to approximate, which is the same as a standard autoencoder.

Autoencoder

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



Let's review the topic.

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

The training examples for an Autoencoder are

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

That is

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

The challenge:

- the input is passed through a "bottleneck" $\backslash\mathbf{z}$ 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

$$\begin{aligned}\hat{\mathbf{R}}^{(d)} &= \beta^{(d)} * \mathbf{F}^{(d)} \\ \hat{\mathbf{R}}^{(d)} &\approx \mathbf{R}^{(d)}\end{aligned}$$

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 $\backslash \mathbf{X}$
  $: (n_{\text{dates}}$
  $\times n_{\text{tickers}}$
  $\times n_{\text{chars}})$
- 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 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 $\beta_s^{(d)} = \text{NN}_\beta(\backslash\mathbf{X}_s^{(d)}; \backslash\mathbf{W}_\beta)$

- $\backslash\mathbf{X}_s^{(d)}$ as input
- parameterized by weights $\backslash\mathbf{W}_\beta$
- $\beta_s^{(d)}$ is only a function of $\backslash\mathbf{X}_s^{(d)}$, the characteristics of s
    - and **not** of any other ticker $s' \neq s$
    - $\beta_s^{(d)}$ shares $\backslash\mathbf{W}_\beta$ 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 $s' \neq s$

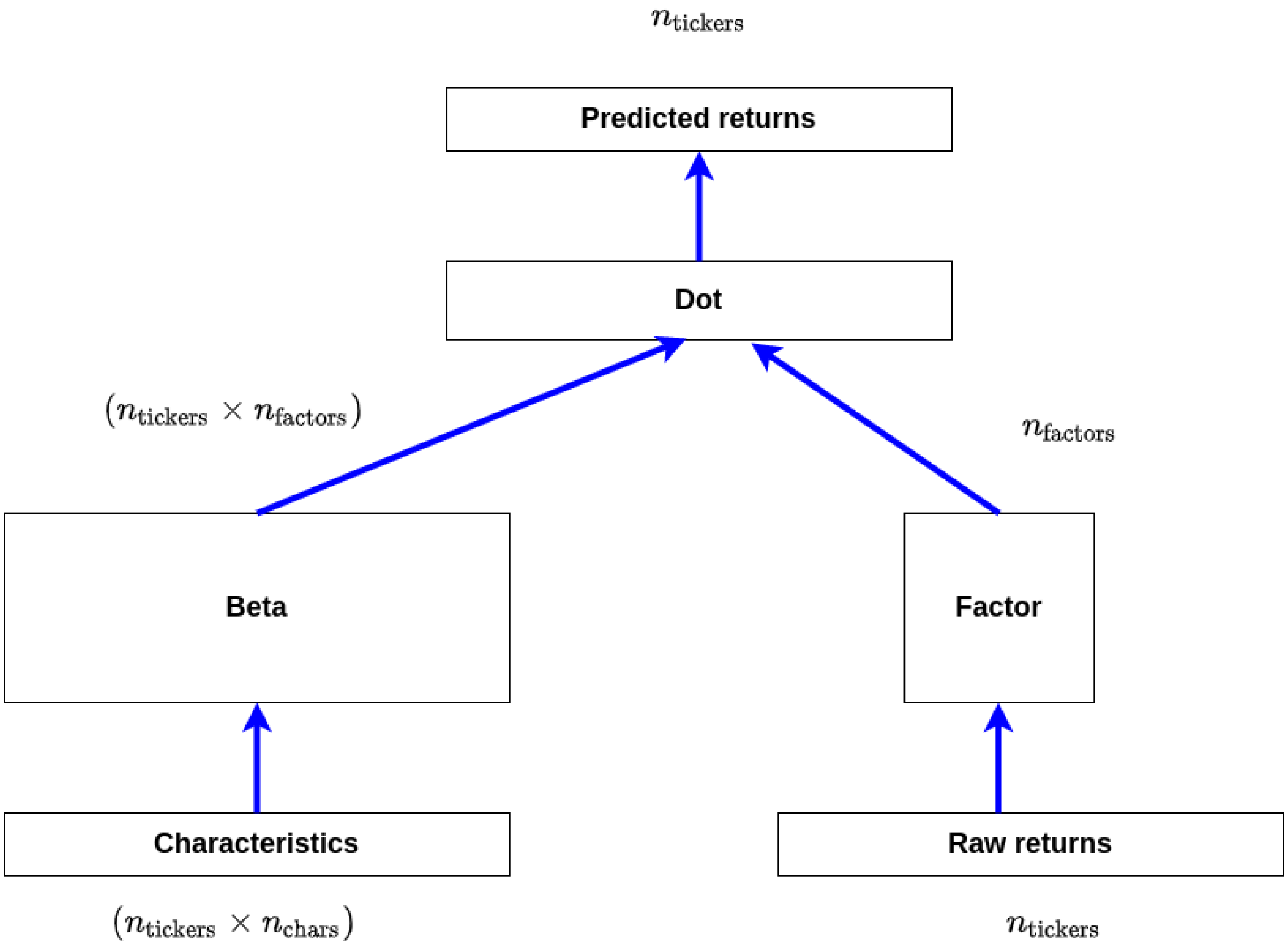
The *Factor network* computes $\mathbf{F}^{(d)} = \mathrm{NN}_{\mathbf{F}}(\mathbf{R}^{(d)}, \backslash\mathrm{W}_{\mathbf{F}})$

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


n_{tickers}
Predicted returns
Dot
$(n_{\text{tickers}} \times n_{\text{factors}})$
n_{factors}
Beta
Factor
Characteristics
Raw returns
$(n_{\text{tickers}} \times n_{\text{chars}})$
n_{tickers}

Summary of this paper

Approximate cross section of daily returns: $\hat{\mathbf{r}}^{(d)} \approx \mathbf{r}^{(d)}$

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

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

Shapes:

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

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 $\backslash\mathbf{X}$

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

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

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

Sub Neural network NN_β

$$\mathrm{NN}_\beta = \texttt{Dense}\ (n_{\text{factors}})(\backslash \mathrm{X})$$

- Fully connected network
- `Dense` (n_{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 `Dense` (n_{factors}) **not** n_{tickers} copies of `Dense`(n_{factors}) with independent weights

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

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

$$\beta^{(d)} = \text{Dense}\left(n_{\text{factors}}, \text{activation='relu'}\right)\left(\backslash \mathbf{X}^{(d)}\right)$$

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

Note that the Beta network

- uses non-linearities (ReLU activation for the `Dense` hidden layer)
- more complex relationship
    - translating the "characteristics" (given as input) of a ticker
    - to its betas with respect to the constructed 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 $\mathbf{NN_F}$

$\mathrm{NN}_{\mathbf{F}} = \texttt{Dense}\ (n_{\mathrm{factors}})$

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

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

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

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

Note that the Factor Network

- **does not** use non-linearities (no activation function in the `Dense` layer)
- design choice
    - factors are thus linear combinations of the input series
    - so can construct the factors as portfolios of the input series
        - with quantities give by weights of the `Dense` network

Dot

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

- this is the predicted return

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

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 $\beta^{(d)}$

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

Loss

The key to any NN is the Loss Function.

Let $\backslash\text{loss}^{(d)}_{(s)}$ denote error of ticker s on day d.

$$\backslash\text{loss}^{(d)}_{(s)} = \mathbf{r}^{(d)}_s - \hat{\mathbf{r}}^{(d)}_s$$

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

$$\backslash\text{loss}^{(d)} = \sum_s \backslash\text{loss}^{(d)}_{(s)}$$

The number of examples m equals n_{dates}

So the Total Loss is

$$\backslash\text{loss} = \sum_d \backslash\text{loss}^{(d)}$$

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

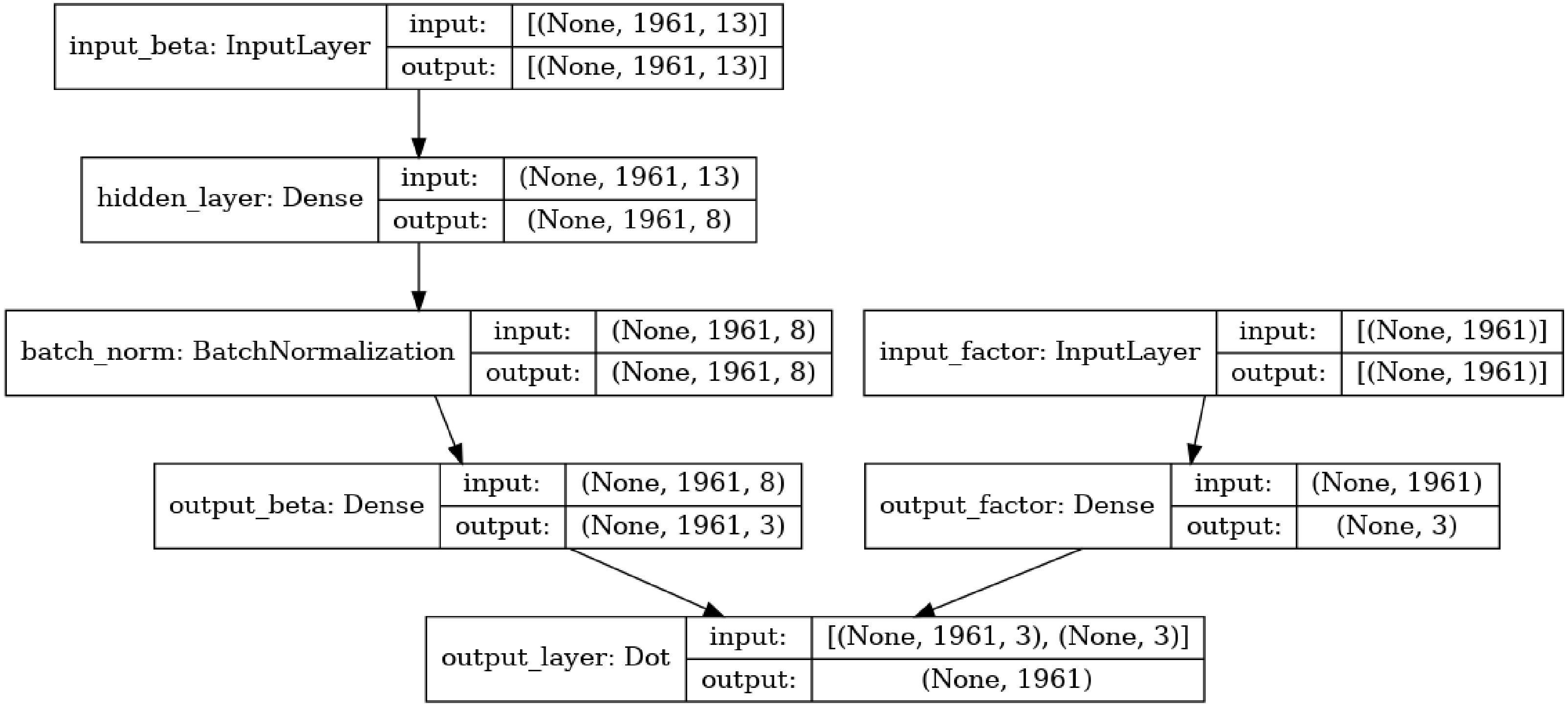
$$\backslash\text{loss}^{(d)}_{(s)} = \mathbf{r}_s^{(d+1)} - \hat{\mathbf{r}}_s^{(d)}$$

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_la
yer')(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)](06_conditional_autoencoder_for_asset_pricing_model.ipynb#Trai) 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 chacteristics

Discussion

Comparison to other factor models

Recall the pre-defined sensitivities factor model

- given sensitivities of tickers to factors
- solve for factors

The sensitivities were not time-varying

- but the cross-sectional regression would trivially accept time-varying sensitivities
- as does the Autoencoder (time-varying "characteristics" per ticker)

The Autoencoder model is most similar to the pre-defined sensitivities model.

The main difference

- pre-defined sensitivities are with respect to factors whose "meaning" has been pre-defined
    - e.g., a "size" or "industry" factor
    - the cross-sectional regression solves for a factor with a *pre-defined* meaning
- Autoencoder model has sensitivities to "characteristics" rather than factors with pre-defined meaning
    - we *solve* for betas with respect to *implied* factors
        - and simultaneously solve for the factor returns
    - the "meaning" of the factors is **not** pre-defined
        - is a function of characteristics
        - determined by weights of the Beta network

Also note that the depth of the Beta network could be greater

- non-linearities at each layer
- more complex relationships

In contrast, the Factor network

- is constructed as a single layer with no activations
- in order to make construction of "factor portfolios" possible

The inputs to the Factor network could be *any* timeseries

- does not have to be the ticker returns
- e.g., liquid, investable instruments such as ETF's

In [6]:

```
print("Done")
```

Done