Public Disclosure Authorized

Public Disclosure Authorized

Policy Research Working Paper 8790

Estimation of the ex ante Distribution of Returns for a Portfolio of U.S. Treasury Securities via Deep Learning

Andrea Foresti

WORLD BANK GROUP
Market and Counterparty Risk Team
March 2019

Policy Research Working Paper 8790

Abstract

This paper presents different deep neural network architectures designed to forecast the distribution of returns on a portfolio of U.S. Treasury securities. A long short-term memory model and a convolutional neural network are tested as the main building blocks of each architecture. The models are then augmented by cross-sectional data and the portfolio's empirical distribution. The paper also presents the fit and generalization potential of each approach.

This paper is a product of the Market and Counterparty Risk Team. It is part of a larger effort by the World Bank to provide open access to its research and make a contribution to development policy discussions around the world. Policy Research Working Papers are also posted on the Web at http://www.worldbank.org/research. The author may be contacted at aforesti@worldbank.org.

The Policy Research Working Paper Series disseminates the findings of work in progress to encourage the exchange of ideas about development issues. An objective of the series is to get the findings out quickly, even if the presentations are less than fully polished. The papers carry the names of the authors and should be cited accordingly. The findings, interpretations, and conclusions expressed in this paper are entirely those of the authors. They do not necessarily represent the views of the International Bank for Reconstruction and Development/World Bank and its affiliated organizations, or those of the Executive Directors of the World Bank or the governments they represent.

Produced by the Research Support Team

Estimation of the ex ante Distribution of Returns for a Portfolio of U.S. Treasury Securities via Deep Learning

Andrea Foresti
The World Bank

JEL: C45, C58, G17
Keywords: Machine Learning, Neural Networks, Convolution, LSTM, Market Risk

1 Introduction

Deep learning is a branch of machine learning that may be loosely defined as a collection of statistical algorithms which feature different layers of operations (usually more than three). These algorithms can be used in either a supervised or unsupervised manner.
Deep learning methods are being actively explored for a variety of practical purposes and have, in recent years, found considerable success. In particular, these methods can be used for both regression (e.g. see [Lathuilière et al., 2018]) and classification tasks (e.g. see [Krizhevsky et al., 2012]). Both these fields of application are very significant in the context of finance.

Moreover, deep learning is of particular interest in all those tasks that present complex input and output data, as the data coming into the model, flowing from one layer to another and finally exiting as output can be arranged in complex multi-dimensional structures.
As we will later see, for instance, classification tasks often entail the specification of the level of confidence across the available classes, thus allowing to give a more nuanced answer than a simple binary (yes/no) attribution.
As a result of all these advantages, in the past few years Deep Learning has started to see fervent research and actual application in finance (as, for instance, recognized in [Board, 2017]).
As an illustration, we have so far seen examples in:

- time series forecasting (e.g. [Sun et al., 2018])
- portfolio construction (e.g. [Heaton et al., 2017])
- credit analysis (e.g. for default prediction see [Hosaka, 2019] or [Hamori et al., 2018])

There has been, however, limited publicly available research in the field of Market Risk.

2 Deep neural networks

A deep neural network is a deep learning model composed of several layers of computations that are loosely inspired by the neurological structure of the brain.
The layers consist of different components (neurons) that are connected to the output of the previous layer according to some pattern. Every layer enacts a transformation on both the content and the shape of the data that flow through it.
The oldest and most common type of layer is the so called *dense* or *fully connected* layer. This type of layer transforms the data by multiplying their inputs by a weight matrix (and adding - optionally - a fixed number (called the *bias*) to its results):

$$X_l = X_{l-1} \cdot W_{l-1}^l + b_l \tag{1}$$

where x_l is the l^th layer in the network, W_{l-1}^l is the weights matrix that transform the data from layer $l-1$ to layer l and b_l is the bias vector. From equation 1 we see that only one bias is applied to every layer, while the weight matrix size is determined by the input and desired output data dimensions.
The output data of a layer are then usually transformed by a function operating element-wise (called the *activation function*).
The most commonly used activation functions are:

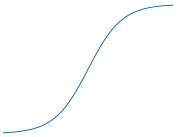
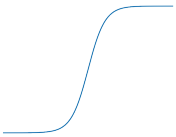
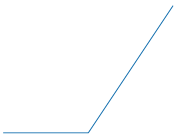
Binary		$\begin{cases} 0 \text{ for } x < 0 \\ 1 \text{ for } x >= 0 \end{cases}$
Sigmoid		$\frac{1}{1+e^{-x}}$
TanH		$\frac{e^x - e^{-x}}{e^x + e^{-x}}$
ReLu		$\begin{cases} 0 \text{ for } x < 0 \\ x \text{ for } x >= 0 \end{cases}$

Table 1: Common activations

When the outputs of a layer are instead directly passed to the following layers, this is normally referred to as a *linear activation*.
What is interesting is that both continuous and discontinuous activation functions are attractive for different purposes during learning.
The fully connected layer works very well for a variety of different tasks but is poorly suited to the analysis of data that are structured in multiple dimensions (such as time-dependent multivariate variables).
While, in theory, a deep learning neural network comprised only of fully connected layers would be able to tackle such a problem, it would in reality present two main problems:

1. in order to be able to learn effectively, the model would need to be of a considerable size (at a minimum hundreds of thousands of parameters);

2. such a model design would have a tendency to overfit the training data.

The first obstacle has been successfully tackled by recent research developments (starting with Geoffrey Hinton's seminal paper [Hinton et al., 2006]) and the availability of ever-greater computing power. The second one, however, has prompted research into different learning architectures that are better suited to extract meta-features from the input data.

One particular solution has been to introduce specialized types of layers to work alongside the fully connected ones.

2.1 Recurrent neural networks and LSTM

A first approach to model time-dependent factors is to use a so-called recurrent neural network (RNN). Recurrent neural networks have the ability to consider time-dependent data, as they work in a way that allows the same set of weights to be applied sequentially to all the items contained in a sample.
RNNs have traditionally been very hard to train and thus they were not used extensively. This changed with the introduction of the Long Short-Term Memory (LSTM) model [Hochreiter and Schmidhuber, 1997]. The LSTM model has, for instance, proven very successful in natural language processing applications and spurred a lot of applications that entail, for instance, text prediction.

The center of the LSTM model is the LSTM cell. A variable number of these cells comprise an LSTM layer, either working in parallel or stacked sequentially.

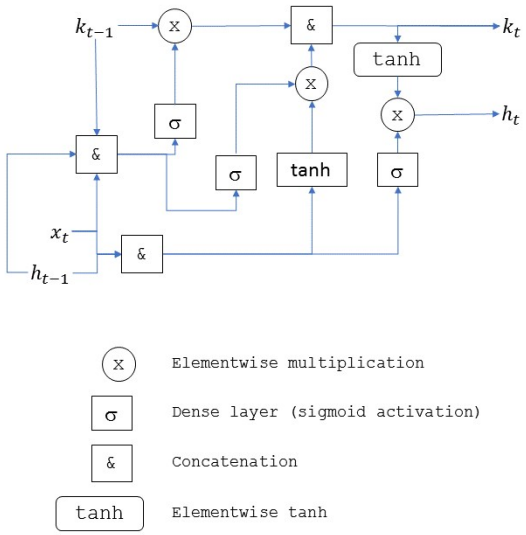


Figure 1: LSTM model cell

We can clearly see in Figure 1 that the LSTM model cell takes both the current item in the data sequence (x_t, which can itself be a vector) and the result of the previous computation on the preceding data item (k_{t-1}) as inputs. There are also some additional data, the *hidden state* of the model, that are computed and passed on to the next element of the data sequence. The ability of the LSTM model to continually update its state across the input sequence is what allows it to better fit sequential data.

2.2 Convolution

A different approach, the use of convolutions, has proven very successful in the field of computer vision and especially object recognition. The use of convolutional neural networks to analyze multivariate time-dependent data appears enticing since these two domains (images and financial time-series data) share similarly shaped data (2D - potentially stacked). In both cases, the data are susceptible to containing meta-features embedded in the interdependencies across the x and y dimensions.
A convolution operation, in the context of deep learning, is defined as the repeated application of a filter (kernel) with fixed weights to different areas of the input data.
More precisely, the output of a convolution operation with kernel size *(R,C)* and stride S is defined as:

$$Output_{i,j} = \sum_{r=0}^{R} \sum_{c=0}^{C} Input_{i \cdot S+r, j \cdot S} + c \cdot Filter_{r,c} \tag{2}$$

Convolutions are particularly powerful when used in sequence, as they can achieve a significant dimensionality reduction on the data, while automatically identifying and extracting the most significant features (see e.g. [Krizhevsky et al., 2012]).
In particular, there is ample evidence of the effectiveness of using several convolutions paired with data pooling (i.e. applying a filter that iterates on the data similarly to a convolution, but that reduces the data in each window to a scalar using a function - such as the average or maximum value).
Another advantage of Convolutional Neural Networks is that they are faster to train than regular feed-forward networks composed of solely fully connected layers. On the other hand, their power often translates in a tendency to overfit the data (i.e. fit the noise in the training data set while having poor performance in the out-of-sample portion of the data). However, several solutions to the problem of overfitting have been proposed, including:

- better feature selection;
- expansion of the number of test cases;
- input data normalization;
- parameter regularization;
- dropout;
- batch normalization.

2.3 Loss functions

The models presented in this paper all belong to the family of supervised learning methods. This means that the parameters of the model are determined as to minimize a specific error ('loss') quantity.
The loss function allows to quantify this error in terms of the distance between a value estimated by the model and the actual ('true') value recorded for the observation.
Common functions used in Machine Learning are:

- Mean Square Error (L2 loss)
- Mean Absolute Error (L1 loss)
- Hinge Loss (Maximum margin loss)
- Cross Entropy Loss

The actual function used depends on the particular problem being studied. For instance, the first two losses are used in regression tasks whereas the last two are used in classification models.

2.4 Data and parameter regularization

Input data are routinely scaled in deep learning in order to achieve faster and better convergence of the model weights. This technique is effectively used to prevent model underfit (as opposed to parameter regularization which aims to achieve the opposite effect).
It is, for instance, common practice either to use either *MinMax* scaling or *Standard* scaling. In the first case, the data are rescaled between two arbitrary values (usually zero and one). In the second case, the data are rescaled to have zero mean and unit variance.

2.4.1 L1 and L2 norms

When the problem, on the other hand is model overfit, kernel weights, biases and layer outputs are instead regularized. A common way to do that is via L_1 and L_2 norms. These are added to the penalty function (scaled by an arbitrary parameter λ) in order to discourage their target from becoming too big.
More precisely (given a vector β of parameters):

$$\begin{aligned} L_1 &= Loss(\beta) + \|\beta\|_1 = Loss(\beta) + \lambda \cdot |\beta| \\ L_2 &= Loss(\beta) + \|\beta\|_2^2 = Loss(\beta) + \lambda \cdot \textstyle\sum_{i=0}^{n} \beta_i^2 \end{aligned} \tag{3}$$

2.4.2 Batch normalization

Batch normalization (see [Ioffe and Szegedy, 2015]) works by normalizing (i.e. subtracting the batch mean and dividing by the batch standard deviation) a layer's activations using their recorded values during each mini-batch. It is another method that has shown results in allowing greater model generalization.

2.4.3 Dropout

Introducing node dropout is yet another method used to achieve better model robustness. In this case a random percentage p of outputs for a pre-determined layer is selected at every gradient update and the values for those nodes is set to zero.
In order to achieve the best effectiveness the dropout percentage p is normally set to a value around 0.5.

3 An application of deep learning on a portfolio of U.S. Treasury bonds

I here analyze the ability of several deep learning architectures to accurately predict the future distribution of a linear portfolio of US Treasury exposures. Such a problem is of particular interest in the field of Market Risk management.
It has to be noted here that a classical value-at-risk (VaR) analysis is not immediately amenable to be performed using the types of models here presented. This is because the use of classical model backtesting performance (i.e. distance between predicted and actual breaches over the observations) as a loss function during training would invariably lead the optimizer to the trivial scalar solution.
In fact, if we define VaR for a variable X as:

$$VaR_\alpha(X) = inf\{x \in \mathbb{R} : F_X(x) > \alpha\} \tag{4}$$

and I backtest over a set of training data using the actual portfolio returns, we see that a model that consistently predicted the $n \cdot \alpha$ worst return would indeed achieve a perfect score.
In other words, the model would not learn from the data instead moving to the (fixed) solution that corresponds to the number of breaches that should be observed during training. This clearly would not have any generalization value. For this reason, it is a lot more appealing to consider the whole future distribution, instead of a single statistic calculated on it.

3.1 Portfolio

The portfolio weights for this analysis are fixed in advance as follows:

$3M$	$6M$	$1Y$	$2Y$	$3Y$	$5Y$	$7Y$	$10Y$	$30Y$
0.1	0.1	0.1	0.3	0.2	0.1	0.05	0.03	0.02

(5)

They will be kept constant across all experiments. A relaxation of this constraint is certainly deserving of future research.

3.2 Market data

I consider 10 years of daily data for US Treasury zero rates (provided by Bloomberg):

Term	Ticker
3M	I02503M Index
6M	I02506M Index
1Y	I0251Y Index
2Y	I0252Y Index
3Y	I0253Y Index
5Y	I0255Y Index
7Y	I0257Y Index
10Y	I02510Y Index
30Y	I02530Y Index

Table 2: Zero rates

Using these rates, I compute the daily returns on the corresponding zero-coupon bonds:

$$r_{t-1,t} = \frac{e^{-r_t \cdot \tau}}{e^{-r_{t-1} \cdot \tau}} \tag{6}$$

These returns are then arranged in a [500 x 9] matrix *for every observation*. In other words, every matrix contains the latest 500 observations of these returns.
The choice of the past 500 observations is motivated by the widely adopted convention in market risk of using the previous two years of returns for the estimation of the parameters used for ex ante risk calculations.

In addition to the return matrix, some model versions presented here augment the returns data with cross-sectional data. These models are referred to in this document as 'mixed' models.
The basic data used in these cases are presented in table 3.

Description	Ticker
US CPI YOY	CPI YOY Index
EU CPI YOY	ECCPEMUY Index
JP CPI YOY	JNCPIYOY Index
EUR 10yr swap rate	EUSA10 Curncy
GBP 10yr swap rate	BPSW10 Curncy
USD 10yr swap rate	USSW10 Curncy
JPY 10yr swap rate	JYSWAP10 Curncy
AUD 10yr swap rate	ADSWAP10 Curncy
3M USD Zero rate	I02503M Index
6M USD Zero rate	I02506M Index
1Y USD Zero rate	I0251Y Index
2Y USD Zero rate	I0252Y Index
3Y USD Zero rate	I0253Y Index
5Y USD Zero rate	I0255Y Index
7Y USD Zero rate	I0257Y Index
10Y USD Zero rate	I02510Y Index
30Y USD Zero rate	I02530Y Index
10Y US Treasury yield	USGG10YR Index
2Y US Treasury yield	USGG2YR Index

Table 3: Base cross-sectional data

In addition, some simple computed features are constructed using the cross-sectional variables as building blocks (Table 4).

	Formula	
10Y US Treasury yield	-	2Y US Treasury yield
USD CPI	avg	50d
USD CPI	avg	100d
USD CPI	avg	200d
3M USD Zero rate	avg	50d
6M USD Zero rate	avg	50d
1Y USD Zero rate	avg	50d
2Y USD Zero rate	avg	50d
3Y USD Zero rate	avg	50d
5Y USD Zero rate	avg	50d
7Y USD Zero rate	avg	50d
10Y USD Zero rate	avg	50d
30Y USD Zero rate	avg	50d

Table 4: Calculated cross-sectional data

The rationale behind pre-computing certain features is to achieve a better economy in model estimation.
Another data source provided to some versions of the models is the empirical (historical) portfolio distribution. It is calculated using the returns matrix associated to each observation to compute a vector of portfolio returns. The histogram of these returns is then used (the portfolio returns having equal weights).
For T observations of a returns vector with members r_t and a vector L $[l_1 \dots l_n]$ of bucket delimiters, the empirical distribution is defined as this vector:

$$H = \begin{bmatrix} h_1 \\ \dots \\ h_{n-1} \end{bmatrix} \text{ where } h_i = \sum_{t=1}^{T} I_i^t \begin{cases} 1 : r_t <= l_{i+1} \cap r_t > l_i \\ 0 : r_t > l_{i+1} \cap r_t <= l_i \end{cases} \tag{7}$$

The buckets are delimited according to this rule:

$$[-\infty; \{\forall i \in [0..20]\, min(R) + (max(R) - min(R))/20 \cdot i\}; \infty] \tag{8}$$

where R is the vector of one-day portfolio returns (defined as: $r_t * positions$ where position is the vector in (5)).

3.3 Target data

The model will try to forecast the distribution of the actual (forward-looking) 10-day returns of the portfolio.

These returns are calculated as:

$$Ret_i^{ptf} = \left(\prod_{t=i+1}^{i+11} \left(1 + r_i^t\right) - 1 \right) \cdot position \tag{9}$$

The returns range is then divided into buckets that are delimited using the same rule defined in equation (8). These buckets can be seen as the classes to which the models will have to assign probabilities.

3.4 Softmax activation

The final layer of all the architectures presented here is a fully connected layer with a *softmax* activation function. This particular activation ouputs a vector whose values sum to one.
The *softmax* transformation of a vector x (with n elements) into an output vector y is defined as:

$$y_i = \frac{e^{x_i}}{\sum_{i=1}^{n} e^{x_i}} \text{ for } i = 1..n \tag{10}$$

The softmax activation function is widely used in classification applications. It allows the model output to be interpreted as the confidence for the state of the world (i.e. future portfolio returns) to belong to any of the available classes.
This means that, in the context of this analysis, the output classes are defined as the discrete bins of the returns distribution.

3.5 Loss function

The loss function used to evaluate the model is the categorical cross-entropy. Over n observations on a model that outputs C classes (i.e. return buckets), entropy (E) is defined in this context as:

$$E = -\sum_{i=1}^{n} \sum_{c=1}^{C} I_c ln(p_i(c))$$

$$\text{with } \begin{cases} I: \text{ Indicator} \begin{cases} 0: \text{ class does not contain actual return} \\ 1: \text{ class contains actual return} \end{cases} \\ p_i(c): \text{ probability assigned by the model on sample i for class c} \end{cases} \tag{11}$$

The choice of this particular loss function results in a loss of zero for a model with perfect foresight while having an infinite value in case a zero probability is wrongly assigned to a class.

3.6 Models

The models were implemented using a Keras 2.2.4 front-end (see [Chollet et al., 2015]) and a Tensorflow 1.12.0 (see [Abadi et al., 2015]) back-end. The first model tested is a simple stacked LSTM model where three LSTM layers operating in sequence are followed by three dense layers:

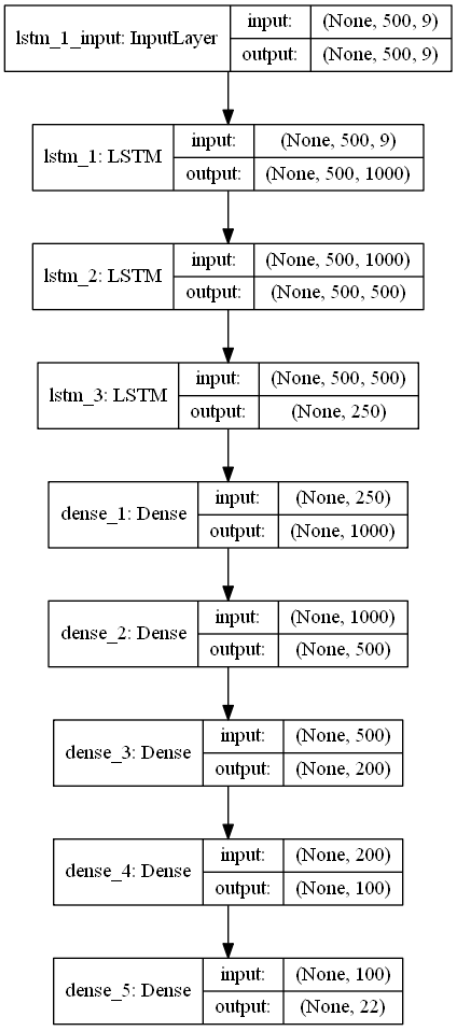


Figure 2: LSTM implementation

The second model is a 2D convolutional model with max pooling. The kernel size for the first convolution is (20,3): this is done to recognize longer-term dependencies in the returns data. The following convolutional layers apply an almost-standard (3,3) convolution

filter.
The convolutions use a 'valid' padding strategy with a unit stride. This results in a fast loss of dimensionality across the returns data.
No regularization was applied.

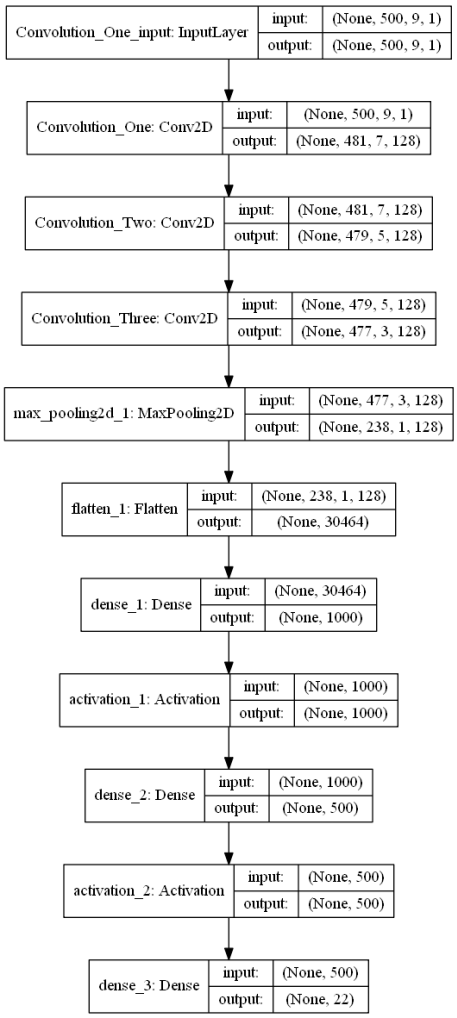


Figure 3: Simple convolution implementation

The third model is an enhanced version of the LSTM model. This iteration employs the original LSTM specification but is augmented by the cross-sectional data that enter through a series of fully connected layers.
The idea behind this extension is to give the model a chance to anchor the predictions to

state data or, in other words, to consider different 'regimes'.

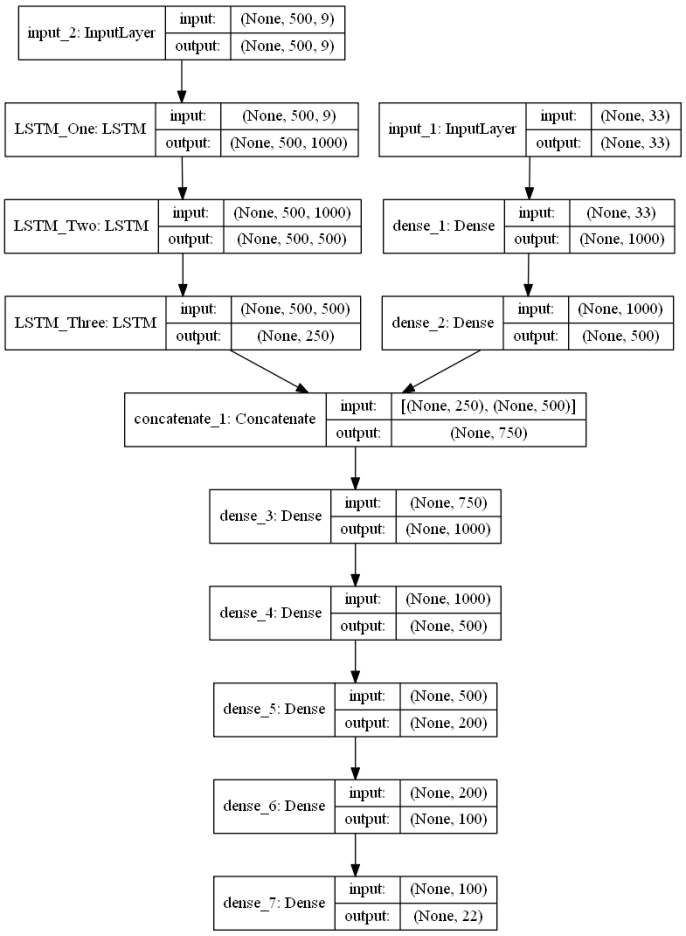


Figure 4: Mixed LSTM implementation

The same augmentation is applied to the original convolutional model.

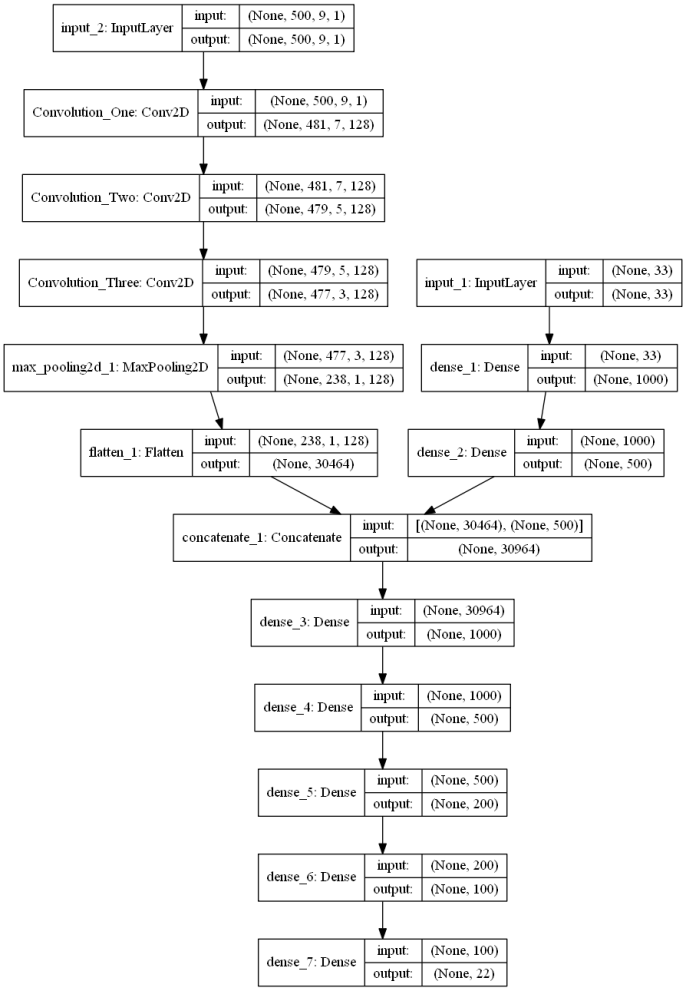


Figure 5: Mixed convolution implementation

The mixed LSTM and convolutional models are then again augmented using the historical distribution of portfolio returns calculated using the past 500 zero-coupon returns available in each observation.
The distribution is segmented using the same buckets used for the output layer and it enters the models through a separate set of dense layers.
When the historical portfolio distribution is added to the mixed LSTM model, I obtain the 'mixed historical LSTM' model (figure 6).

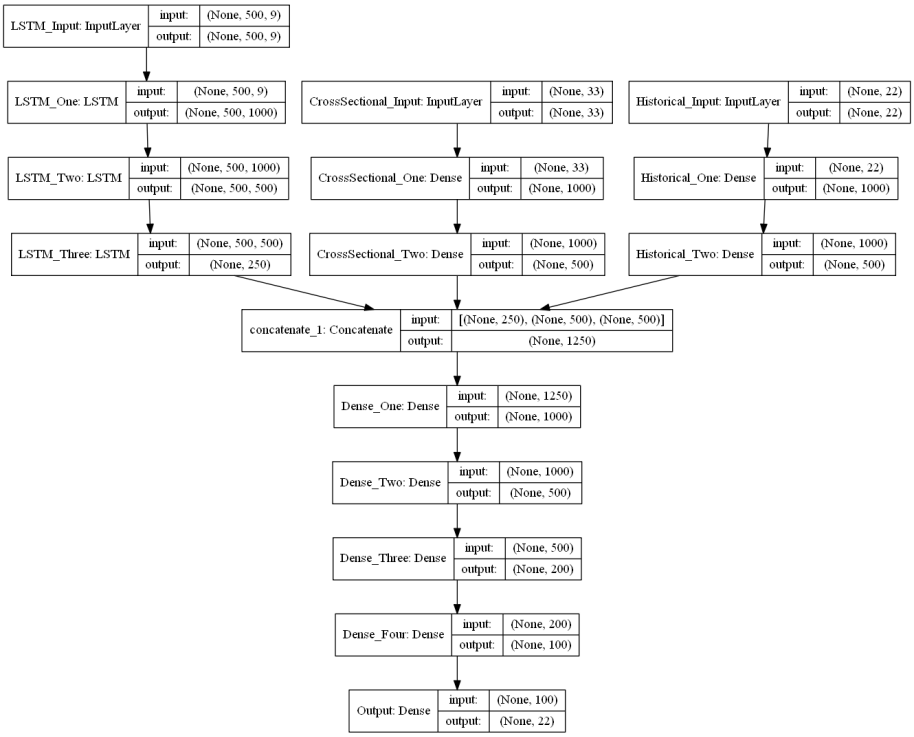


Figure 6: Mixed historical LSTM implementation

The mixed convolutional model becomes the 'mixed historical convolutional' model when the empirical historical portfolio distribution is added to it (figure 7).

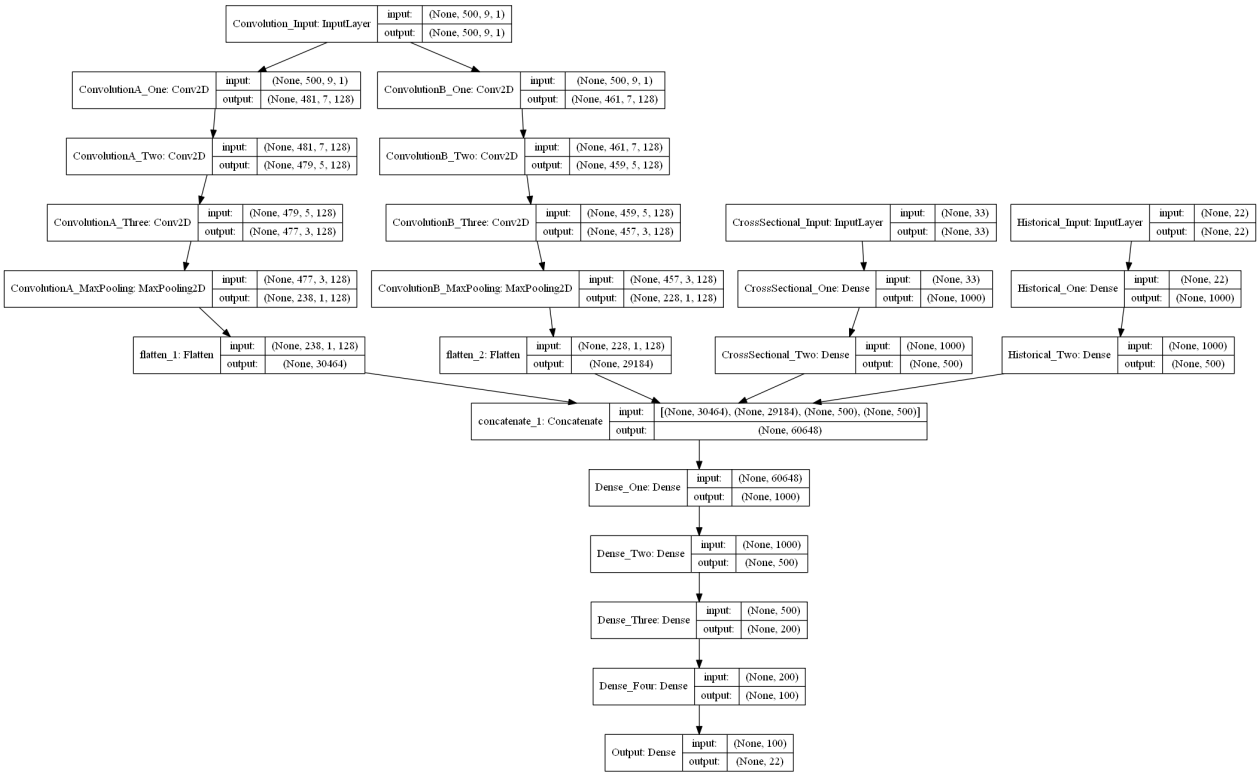


Figure 7: Mixed historical convolutional implementation

The only regularization applied is one pass of the *MinMax* data scaler for the cross-sectional data (when used by the model).

3.7 Benchmark models

In order to better assess the performance of the models introduced here, a couple of reference approaches are also considered.
The first one is a classic multivariate Gaussian parametric model. It is calculated using the return matrix for each observation and computing from it a covariance matrix. An exponential weighting with a decay factor of 0.94 is used for the estimation. The expected return is fixed at zero.
The covariance matrix is used in conjunction with portfolio weights to determine a forward-looking portfolio standard deviation value, thus determining the ex ante portfolio distribution.
The second benchmark used is the historical distribution of portfolio returns described in equation 7.

4 Results

The models are tested using a 0.8/0.2 training/validation data set randomized split. This results in having 2,068 training cases and 517 test cases.
The training data set is further randomized at the beginning of each training epoch.
The optimization was run using an Adam optimizer (with the original parameters introduced in [Kingma and Ba, 2014]) and a minibatch size of 128. Each model was run with a so-called 'patience' parameter of 10 for early termination. This means that the optimization would wait for 10 epochs after the validation loss stopped improving.

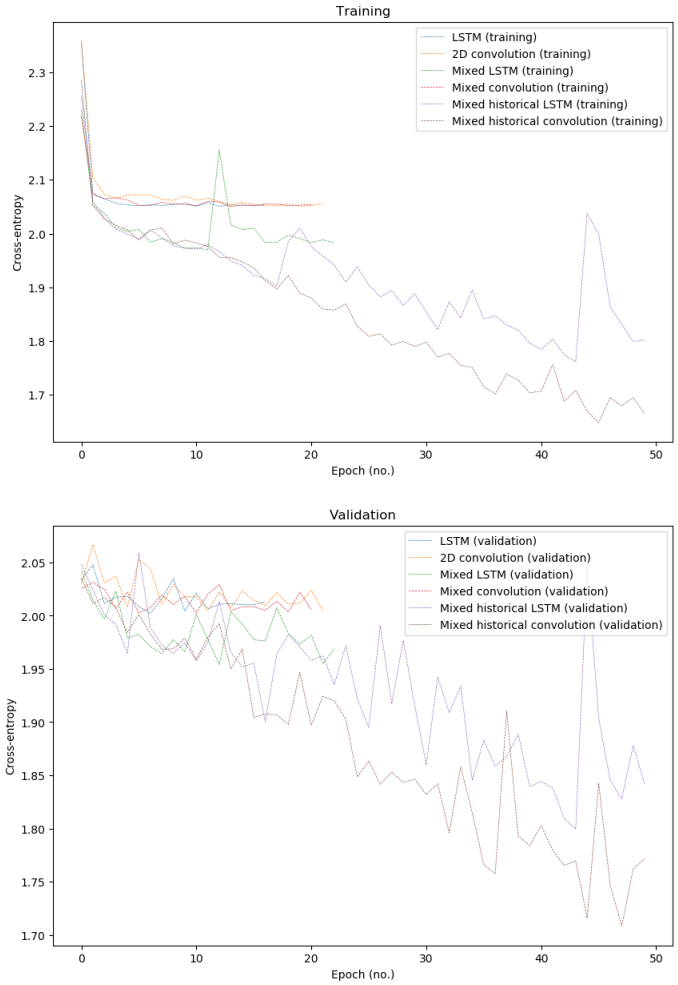


Figure 8: Training and validation loss per epoch

I now choose as the candidate model versions using these criteria (in order of decreasing importance):

- least validation loss
- least training loss
- latest epoch

After training the models presented here, this resulted in choosing these models:

Model	Epoch	Training loss	Validation loss
LSTM	07	2.05	2.00
Convolutional	22	2.05	2.00
Mixed LSTM	22	1.98	1.95
Mixed convolutional	19	2.05	2.00
Mixed historical LSTM	44	1.76	1.80
Mixed historical convolutional	49	1.66	1.76
Parametric	-	2.19	2.17
Historical	-	2.07	2.02

Table 5: Optimal model performance

In order to better judge the models, the shape of the produced distribution is considered. It is indeed very important to have models that produce meaningful distribution shapes and that react to different data.
Machine learning models have a normal tendency to quickly converge onto degenerate solutions that nevertheless have a strong explanatory power (e.g. constant solutions).
This, in a way, could be considered as a special case of overfit.
So, to measure the variability of the distributions produced by each model, I consider the Bhattacharyya distance (introduced in [Bhattacharyya, 1946]):

$$BD = -ln\left(\sum_{x \in X} \sqrt{p(x)q(x)}\right) \tag{12}$$

For every model, I compute the Bhattacharyya distance between the latest available returns distribution and all the other distributions generated over the rest of the data.
The results are illustrated in Figure 9.

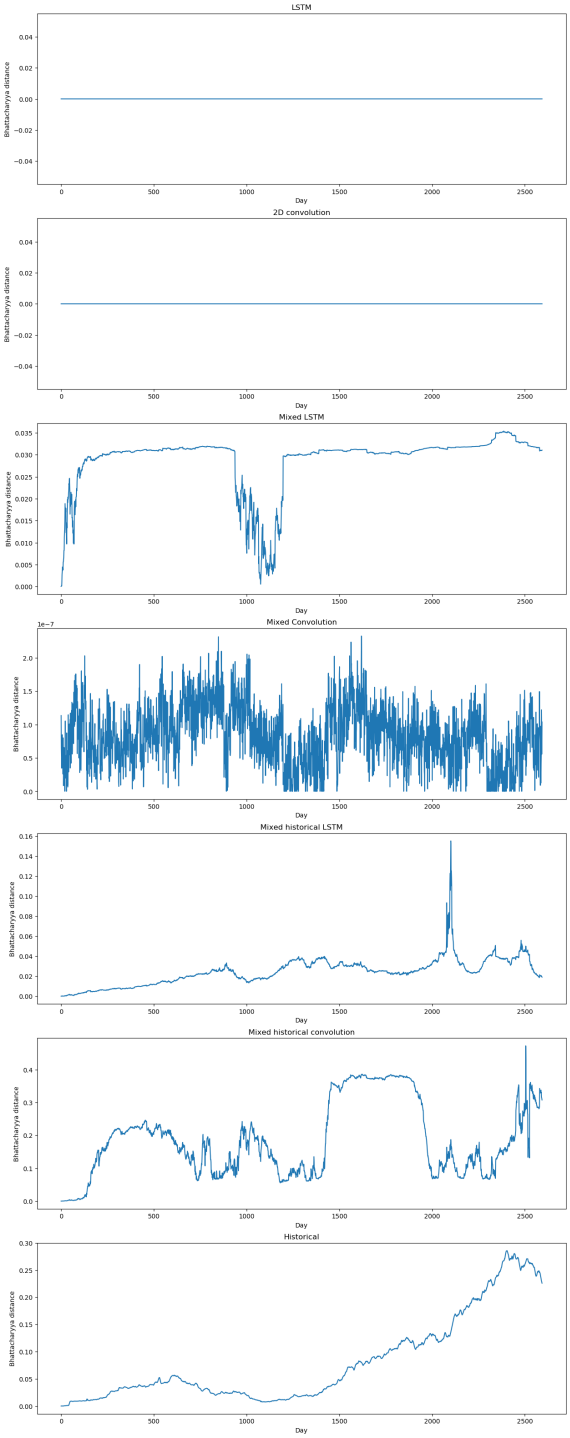


Figure 9: Bhattacharyya distance from latest distribution

In order to offer a better comparison, the same distance is applied also to the empirical portfolio distributions (i.e. the historical model distribution) results.

It is clearly possible to appreciate how the complexity of the model, while maybe not significantly improving the training and validation performance, allows the creation of more reactive models.
This gives us a hint about the possibility that more complex models might offer better generalization power.
In particular, we see that the naive application of both the LSTM and convolutional model result in degenerate behavior of the distance among the predicted distributions.

For the sake of illustration, figure 10 presents the distributions predicted by each model for the latest available observation in the data set.

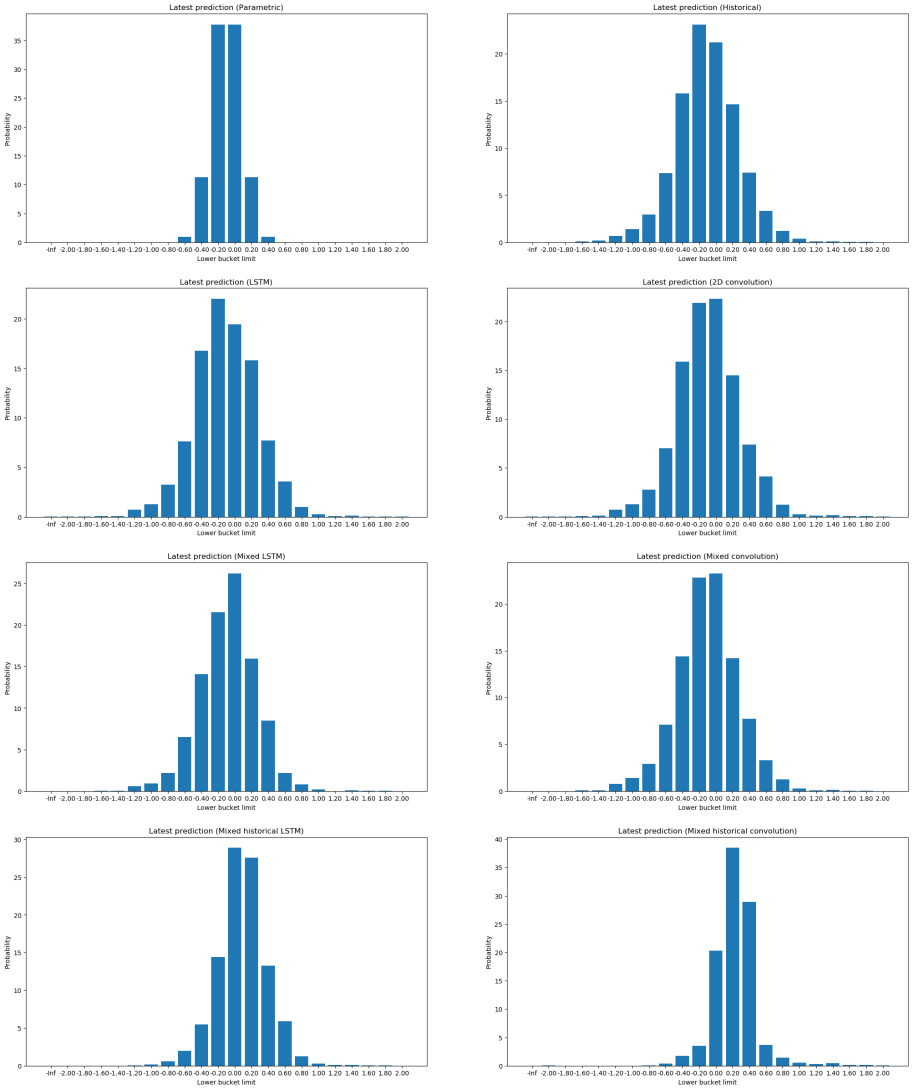


Figure 10: Latest predicted distribution

5 Conclusion

The results I obtained corroborate the notion that the use of deep-learning methods for financial application is viable. In particular, a finding which is consistent with existing deep learning literature is that models need to cross a complexity threshold in order to achieve significant efficacy.
Whether the models here presented already achieved this critical mass is, though, not clear. Furthermore, it has to be underlined that these tools are still poorly understood and much more difficult to calibrate than more traditional methods. It is not immediately apparent which features are more important than others (especially considering the limited nature of these experiments).
Moreover, the long time to convergence for some of the models indicates that more could be done in order to speed up learning.

Finally, these results suggest a few opportunities for further research:

- Introducing a bigger number of factors onto which the portfolio is mapped;
- Testing different model architectures, hyper-parameters and regularization techniques. The use of auto-encoders for dimensionality reduction is, in this context, particularly appealing in light of possible real-world applications (i.e. models featuring hundreds or thousands of different factors);
- Exploring different cross-sectional data features;
- Verifying the soundness of the model(s) for different portfolio structures;
- Investigating the use of different penalty functions that would be more of interest in the field of risk measurement (e.g. weighted cross-entropy).

References

[Abadi et al., 2015] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

[Bhattacharyya, 1946] Bhattacharyya, A. (1946). On a measure of divergence between two multinomial populations. *Sankhyā: the indian journal of statistics*, pages 401–406.

[Board, 2017] Board, F. S. (2017). Artificial intelligence and machine learning in financial services. *November, available at: http://www. fsb. org/2017/11/artificialintelligence-and-machine-learning-in-financialservice/(accessed 30th January, 2018)*.

[Chollet et al., 2015] Chollet, F. et al. (2015). Keras. https://keras.io.

[Hamori et al., 2018] Hamori, S., Kawai, M., Kume, T., Murakami, Y., and Watanabe, C. (2018). Ensemble learning or deep learning? application to default risk analysis. *Journal of Risk and Financial Management*, 11(1):12.

[Heaton et al., 2017] Heaton, J., Polson, N., and Witte, J. H. (2017). Deep learning for finance: deep portfolios. *Applied Stochastic Models in Business and Industry*, 33(1):3–12.

[Hinton et al., 2006] Hinton, G. E., Osindero, S., and Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

[Hosaka, 2019] Hosaka, T. (2019). Bankruptcy prediction using imaged financial ratios and convolutional neural networks. *Expert Systems with Applications*, 117:287–299.

[Ioffe and Szegedy, 2015] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

[Kingma and Ba, 2014] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

[Lathuilière et al., 2018] Lathuilière, S., Mesejo, P., Alameda-Pineda, X., and Horaud, R. (2018). A comprehensive analysis of deep regression. *arXiv preprint arXiv:1803.08450*.

[Sun et al., 2018] Sun, S., Wei, Y., and Wang, S. (2018). Adaboost-lstm ensemble learning for financial time series forecasting. In *International Conference on Computational Science*, pages 590–597. Springer.