Deep Learning for Finance: Deep Portfolios

J. B. Heaton [*] N. G. Polson [†] J. H. Witte [‡]

First Draft: February 2016
This Draft: September 2016

Abstract

We explore the use of deep learning hierarchical models for problems in financial prediction and classification. Financial prediction problems – such as those presented in designing and pricing securities, constructing portfolios, and risk management – often involve large data sets with complex data interactions that currently are difficult or impossible to specify in a full economic model. Applying deep learning methods to these problems can produce more useful results than standard methods in finance. In particular, deep learning can detect and exploit interactions in the data that are, at least currently, invisible to any existing financial economic theory.

Key Words: Deep Learning, Machine Learning, Big Data, Artificial Intelligence, Finance, Asset Pricing, Volatility, Deep Frontier

[*]Conjecture LLC, jb@conjecturellc.com
[†]Booth School of Business, University of Chicago, ngp@chicagobooth.edu
[‡]Department of Mathematics, University College London, and Conjecture LLC, jhw@conjecturellc.com

1 Introduction

Deep learning is a form of machine learning, the use of data to train a model to make predictions from new data. Recent advances in deep learning have improved dramatically the ability of computers to recognize and label images, recognize and translate speech, and play games of skill, in each case at better than human-level performance. In these prominent applications of deep learning, the goal is typically to train a computer to do as well or better than a human some task - such as recognizing the content of an image - that a human is assumed (often implicitly) able to do quite well. Deep learning methods allow the task to be performed without human involvement, perhaps because the task otherwise capably performed by a human must be performed on a scale much larger than the availability of human workers will allow in the time given (such as recognizing the content of new images posted to Facebook or Google, see Dean et al., 2012) or because the benefits of above-human-level performance are extremely high (such as in medical diagnoses).

Problems in the financial markets may be different from typical deep learning applications in many respects, but one of the most notable is that the emphasis is not on replicating tasks that humans already do well. Unlike recognizing an image or responding appropriately to verbal requests, humans have no innate ability to, for example, select a stock that is likely to perform well in some future period. Nevertheless, the tools of deep learning may be useful in such selection problems, since deep learning techniques are, at their core, simply the best available way to compute any function mapping data (which may be other returns, economic data, accounting date, demographic data, data on the legal regime, etc.) into the value of the return. In theory at least, a deep learner can find the relationship for a return, no matter how complex and non-linear. This is a far cry from both the simplistic linear factor models of traditional financial economics and from the relatively crude, ad hoc methods of statistical arbitrage and other quantitative asset management techniques.

The rest of the paper continues as follows. Section 2 introduces our deep learning framework and the training of a deep architecture. Section 3 provides a traditional probabilistic interpretation of the modeling procedure. Section 4 describes our four-step algorithm for model construction and validation with particular emphasis on building deep portfolios. We provide a smart indexing example by auto-encoding the IBB biotechnology index. Section 5 concludes.

2 Deep Learning

The fundamental machine learning problem is to find a predictor of an output Y given an input X. A learning machine is an input-output mapping, $Y = F(X)$, where the input space is high-dimensional and we write

$$Y = F(X) \text{ where } X = (X_1, \ldots, X_p).$$

The output Y can be continuous, discrete as in classification, or mixed (see also Ripley, 1996). For example, in a classification problem, we need to learn a mapping $F : X \to Y$, where $Y \in \{1, \ldots, K\}$ indexes categories. A predictor is denoted by $\hat{Y}(X) := F(X)$.

As a form of machine learning, deep learning trains a model on data to make predictions, but is distinguished by passing learned features of data through different *layers* of abstraction. The deep

approach employs hierarchical predictors comprising of a series of L non-linear transformations applied to X. Each of the L transformations is referred to as a *layer*, where the original input is X, the output of the first transformation is the first layer, and so on, with the output $\hat{Y}$ as the $(L+1)$-th layer. We use $l \in \{1, \ldots, L\}$ to index the layers from 1 to L, which are called *hidden layers*. The number of layers L represents the *depth* of our architecture.

Specifically, a deep learning architecture can be described as follows. Let $f_1, \ldots, f_L$ be given univariate activation functions for each of the L layers. Activation functions are non-linear transformations of weighted data. Commonly used activation functions are sigmoidal (e.g., $1/(1+\exp(-x))$, $\cosh(x)$, or $\tanh(x)$), heaviside gate functions (e.g., $\mathbb{I}(x > 0)$), or rectified linear units (ReLU) $\max\{x, 0\}$. A semi-affine activation rule is then defined by

$$f_l^{W,b} := f_l \left(\sum_{j=1}^{N_l} W_{lj} X_j + b_l \right) = f_l(W_l X_l + b_l), \quad 1 \leq l \leq L,$$

which implicitly needs the specification of the number of hidden units N_l. Our deep predictor, given the number of layers L, then becomes the composite map

$$\hat{Y}(X) := F(X) = \left(f_1^{W_1,b_1} \circ \ldots \circ f_L^{W_L,b_L} \right)(X).$$

Put simply, we model a high dimensional mapping, F, via the concatenation of univariate semi-affine functions (Kolmogorov, 1957). Similar to a classic basis decomposition (e.g., see Diaconis et al., 1984), the deep approach uses univariate activation functions to decompose a high dimensional X (see also Lorentz, 1976, Gallant et al., 1988, Poggio et al., 1990, and Hornik et al., 1989).

We let $Z^{(l)}$ denote the l-th layer, and so $X = Z^{(0)}$. The final output is the response Y, which can be numeric or categorical. The explicit structure of a deep prediction rule is then

$$\begin{aligned}
Z^{(1)} &= f^{(1)} \left(W^{(0)} X + b^{(0)} \right), \\
Z^{(2)} &= f^{(2)} \left(W^{(1)} Z^{(1)} + b^{(1)} \right), \\
&\ldots \\
Z^{(L)} &= f^{(L)} \left(W^{(L-1)} Z^{(L-1)} + b^{(L-1)} \right), \\
\hat{Y}(X) &= W^{(L)} Z^{(L)} + b^{(L)}.
\end{aligned}$$

Here, $W^{(l)}$ are weight matrices, and $b^{(l)}$ are threshold or activation levels. Designing a good predictor depends crucially on the choice of univariate activation functions $f^{(l)}$. The $Z^{(l)}$ are hidden features (or factors) which the algorithm extracts. One particular noticeable property is that the weights $W_l \in \mathbb{R}^{N_l \times N_{l-1}}$ are matrices.

This gives the predictor great flexibility to uncover non-linear features of the data – particularly so in finance data, since the estimated hidden features $Z^{(l)}$ can represent portfolios of payouts. The choice of the dimension N_l is key, however, since if a hidden unit (columns of W_l) is dropped at layer l, then it eliminates all terms above it in the layered hierarchy.

It is common to split the data-set into three subsets, namely training, validation, and testing. The training set is used to adjust the weights of the network. The validation set is used to minimize the over-fitting and relates to the architecture design (a.k.a. model selection). Finally, testing is used to confirm the actual predictive power of a learner.

Once the activation functions, size, and depth of the learning routine have been chosen, we need to solve the training problem of finding $(\hat{W}, \hat{b})$, where

$$\hat{W} = (\hat{W}_0, \ldots, \hat{W}_L) \text{ and } \hat{b} = (\hat{b}_0, \ldots, \hat{b}_L)$$

denote the learning parameters which we compute during training. To do this, we need a training dataset $D = \{Y^{(i)}, X^{(i)}\}_{i=1}^{T}$ of input-output pairs and a loss function $\mathcal{L}(Y, \hat{Y})$ at the level of the output signal. In its simplest form, we solve

$$\arg\min_{W,b} \frac{1}{T} \sum_{i=1}^{T} \mathcal{L}(Y_i, \hat{Y}^{W,b}(X_i)) \,. \tag{1}$$

It is common to add a regularization penalty, denoted by $\phi(W, b)$, to avoid over-fitting and to stabilize our predictive rule. We combine this with the loss function via a parameter $\lambda > 0$, which gauges the overall level of regularization. We then need to solve

$$\arg\min_{W,b} \frac{1}{T} \sum_{i=1}^{T} \mathcal{L}(Y_i, \hat{Y}^{W,b}(X_i)) + \lambda\phi(W, b) \,. \tag{2}$$

The choice of the amount of regularization, λ, is a key parameter. This gauges the trade-off present in any statistical modeling that too little regularization will lead to over-fitting and poor out-of-sample performance.

In many cases, we will take a separable penalty, $\phi(W, b) = \phi(W) + \phi(b)$. The most useful penalty is the ridge or L^2-norm, which can be viewed as a default choice, namely

$$\phi(W) = \|W\|_2^2 = \sum_{i=1}^{T} W_i^\top W_i.$$

Other norms include the lasso, which corresponds to an L^1-norm, and which can be used to induce sparsity in the weights and/or off-sets. The ridge norm is particularly useful when the amount of regularization, λ, has itself to be learned. This is due to the fact that there are many good predictive generalization results for ridge-type predictors. When sparsity in the weights is paramount, it is common to use a lasso L^1-norm penalty.

The common numerical approach for the solution of (2) is a form of stochastic gradient descent (SGD), which adapted to a deep learning setting is usually called *back-propagation*. One caveat of back-propagation in this context is the multi-modality of the system to be solved (and the resulting slow convergence properties), which is the main reason why deep learning methods heavily rely on the availability of large computational power.

One of the advantages of using a deep network is that first-order derivative information is directly

available. There are tensor libraries available that directly calculate

$$\nabla_{W,b}\mathcal{L}(Y_i, \hat{Y}^{W,b}(X_i))$$

using the chain rule across the training data-set. For ultra-large data-sets, we use mini-batches and SGD to perform this optimization, see LeCun et al. (2012). An active area of research is the use of this information within a Langevin MCMC algorithm that allows sampling from the full posterior distribution of the architecture. The deep learning model by its very design is highly multi-modal, and the parameters are high dimensional and in many cases unidentified in the traditional sense. Traversing the objective function is the desired problem, and handling the multi-modal and slow convergence of traditional decent methods can be alleviated with proximal algorithms such as the alternating method of multipliers (ADMM), as has been discussed in Polson et al. (2015).

There are two key training problems that can be addressed using the predictive performance of an architecture.

(i) How much regularization to add to the loss function. As indicated before, one approach is to use cross validation and to teach the algorithm to calibrate itself to a training data. An independent hold-out data set is kept separately to perform an out-of-sample measurement of the training success in a second step. As we vary the amount of regularization, we obtain a regularization path and choose the level of regularization to optimize out-of-sample predictive loss. Another approach is to use Stein's unbiased estimator of risk (SURE, Stein, 1981).

(ii) A more challenging problem is to train the size and depth of each layer of the architecture, i.e., to determine L and $N = (N_1, \ldots, N_L)$. This is known as the model selection problem. In the next subsection, we will describe a technique known as dropout, which solves this problem.

Stein's unbiased estimator of risk (SURE) proceeds as follows. For a stable predictor, $\hat{Y}$, we can define the degrees of freedom of a predictor by $df = \mathbb{E}\left(\sum_{i=1}^{T} \partial\hat{Y}_i/\partial Y_i\right)$. Then, given the scalability of our algorithm, the derivative $\partial\hat{Y}/\partial Y$ is available using the chain rule for the composition of the L layers.

Now let the in-sample MSE be given by $err = ||Y - \hat{Y}||_2^2$ and, for a future observation $Y^\star$, the out-of-sample predictive MSE is

$$Err = \mathbb{E}_{Y^\star}\left(||Y^\star - \hat{Y}||_2^2\right).$$

In expectation, we then have $\mathbb{E}\left(Err\right) = \mathbb{E}\left(err + 2\mathrm{Var}(\hat{Y}, Y)\right)$ where the expectation is taken over the data generating process. The latter term is a covariance and depends on df. Stein's unbiased risk estimate then becomes

$$\widehat{\mathrm{Err}} = ||Y - \hat{Y}||^2 + 2\sigma^2 \sum_{i=1}^{n} \frac{\partial\hat{Y}_i}{\partial Y_i}.$$

Models with the best predictive MSE are favoured.

Dropout is a model selection technique. It is designed to avoid over-fitting in the training process,

and does so by removing input dimensions in X randomly with a given probability p. In a simple model with one hidden layer, we replace the network

$$\begin{aligned} Y_i^{(l)} &= f(Z_i^{(l)}), \\ Z_i^{(l)} &= W_i^{(l)} X^{(l)} + b_i^{(l)}, \end{aligned}$$

with the dropout architecture

$$\begin{aligned} D_i^{(l)} &\sim \text{Ber}(p), \\ \tilde{Y}_i^{(l)} &= D^{(l)} \star X^{(l)}, \\ Y_i^{(l)} &= f(Z_i^{(l)}), \\ Z_i^{(l)} &= W_i^{(l)} X^{(l)} + b_i^{(l)}. \end{aligned}$$

In effect, this replaces the input X by $D \star X$, where $\star$ denotes the element-wise product and D is a matrix of independent Bernoulli Ber(p) distributed random variables.

It is instructive to see how this affects the underlying loss function and optimization problem. For example, suppose that we wish to minimise MSE, $\mathcal{L}(Y, \hat{Y}) = \|Y - \hat{Y}\|_2^2$, then, when marginalizing over the randomness, we have a new objective

$$\arg\min_W \mathbb{E}_{D \sim \text{Ber}(p)} \|Y - W(D \star X)\|_2^2 \,,$$

With $\Gamma = (\text{diag}(X^\top X))^{\frac{1}{2}}$, this is equivalent to

$$\arg\min_W \|Y - pWX\|_2^2 + p(1-p)\|\Gamma W\|_2^2 \,,$$

We can also interpret the last expression as a Bayesian ridge regression with a g-prior. Put simply, dropout reduces the likelihood of over-reliance on small sets of input data in training. See Hinton and Salakhutdinov (2006) and Srivastava et al. (2014). Dropout can be viewed as the optimization version of model selection. This contrasts with the traditional spike-and-slab prior (that has proven so popular in Bayesian model-averaging) which switches between probability models and requires computationally intensive MCMC models for implementation.

Another application of dropout regularization is the choice of the number of hidden units in a layer. This can be achieved if we drop units of the hidden rather than the input layer and then establish which probability p gives the best results.

3 Probabilistic Interpretation

In a traditional probabilistic setting, we could view the output Y as a random variable generated by a probability model $p(Y|Y^{W,b}(X))$, where the conditioning is on the predictor $\hat{Y}(X)$. The corresponding loss function is then

$$\mathcal{L}(Y, \hat{Y}) = -\log p(Y|Y^{\hat{W},\hat{b}}(X)),$$

namely the negative log-likelihood. For example, when predicting the probability of default, we have a multinomial logistic regression model which leads to a cross-entropy loss function. Often, the L_2-norm for a traditional least squares problem

$$\mathcal{L}(Y_i, \hat{Y}(X_i)) = \|Y_i - \hat{Y}(X_i)\|_2^2,$$

is chosen as an error measure, giving a mean-squared error (MSE) target function.

Probabilistically, the regularization term, $\lambda\phi(W, b)$, can be viewed as a negative log-prior distribution over parameters, namely

$$\begin{aligned} -\log p(\phi(W, b)) &= \lambda\phi(W, b), \\ p(\phi(W, b)) &\propto \exp(-\lambda\phi(W, b)). \end{aligned}$$

This framework then provides a correspondence with Bayes learning. Our deep predictor is simply a regularized maximum a posteriori (MAP) estimator. We can show this using Bayes rule as

$$\begin{aligned} p(W, b|D) &\propto p(Y|Y^{W,b}(X))p(W, b) \\ &\propto \exp\Big(-\log p(Y|Y^{W,b}(X)) - \log p(W, b)\Big), \end{aligned}$$

and the deep learning predictor satisfies

$$\hat{Y} := Y^{\hat{W},\hat{b}}(X) \;\text{ where }\; (\hat{W}, \hat{b}) := \arg\min_{W,b} \; \log p(W, b|D),$$

and

$$-\log p(W, b|D) = \sum_{i=1}^{T} \mathcal{L}(Y^{(i)}, Y^{W,b}(X^{(i)})) + \lambda\phi(W, b)$$

is the log-posterior distribution over parameters given the training data, $D = \{Y^{(i)}, X^{(i)}\}_{i=1}^{T}$. (For more detail on the experimental link between deep learning and probability theory, see also Lake et al., 2015).

4 Stacked Auto-Encoders

For finance applications, one of the most useful deep learning applications is an auto-encoder. An auto-encoder is a deep learning routine which trains the architecture to replicate X itself, namely $X = Y$, via a *bottleneck* structure. This means we select a model $F^{W,b}(X)$ which aims to concentrate the information required to recreate X. Put differently, an auto-encoder creates a more cost effective representation of X. Suppose that we have N input vectors $X = \{x_1, \ldots, x_N\} \in \mathbb{R}^{M\times N}$ and N output (or target) vectors $\{x_1, \ldots, x_N\} \in \mathbb{R}^{M\times N}$. If (for simplicity) we set biases to zero and use

one hidden layer ($L = 2$) with only $K < N$ factors, then our input-output *market-map* becomes

$$\begin{aligned} Y_j(x) = F_W^m(X)_j &= \sum_{k=1}^{K} W_2^{jk} f\left(\sum_{i=1}^{N} B_1^{ki} x_i\right) \\ &= \sum_{k=1}^{K} W_2^{jk} Z_j \text{ for } Z_j = f\left(\sum_{i=1}^{N} W_1^{ki} x_i\right), \end{aligned}$$

for $j = 1, \ldots, N$, where $f(\cdot)$ is a univariate activation function.

Since, in an auto-encoder, we are trying to fit the model $X = F_W(X)$, in the simplest possible case, we *train* the weights $W = (W_1, W_2)$ via a criterion function

$$\begin{aligned} \mathcal{L}(W) &= \arg\min_W \|X - F_W(X)\|^2 + \lambda\phi(W) \\ \text{with} \quad \phi(W) &= \sum_{i,j,k} |W_1^{jk}|^2 + |W_2^{ki}|^2, \end{aligned}$$

where λ is a regularization penalty.

If we use an augmented Lagrangian (as in ADMM) and introduce the latent factor Z, then we have a criterion function that consists of two steps, an encoding step (a penalty for Z), and a decoding step for reconstructing the output signal via

$$\arg\min_{W,Z} \|X - W_2 Z\|^2 + \lambda\phi(Z) + \|Z - f(W_1, X)\|^2,$$

where the regularization on W_1 induces a penalty on Z. The last term is the encoder, the first two the decoder.

In an auto-encoder, for a training data set $\{X_1, X_2, \ldots\}$, we set the target values as $Y_i = X_i$. A static auto-encoder with two linear layers, akin to a traditional factor model, can be written as a deep learner as

$$\begin{aligned} z^{(2)} &= W^{(1)} X + b^{(1)}, \\ a^{(2)} &= f_2(z^{(2)}), \\ z^{(3)} &= W^{(2)} a^{(2)} + b^{(2)}, \\ Y = F^{W,b}(X) &= a^{(3)} = f_3(z^{(3)}), \end{aligned}$$

where $a^{(2)}, a^{(3)}$ are activation levels. It is common to set $a^{(1)} = X$. The goal is to learn the weight matrices $W^{(1)}, W^{(2)}$. If $X \in \mathbb{R}^N$, then $W^{(1)} \in \mathbb{R}^{M,N}$ and $W^{(1)} \in \mathbb{R}^{N,M}$, where $M \ll N$ provides the auto-encoding at a lower dimensional level.

If W_2 is estimated from the structure of the training data matrix, then we have a traditional factor model, and the W_1 matrix provides the factor loadings. (We note that PCA in particular falls into this category, see Cook, 2007). If W_2 is estimated based on the pair $\hat{X} = \{Y, X\} = X$ (which means estimation of W_2 based on the structure of the training data matrix with the specific auto-encoder objective), then we have a sliced inverse regression model. If W_1 and W_2 are simultaneously estimated based on the training data X, then we have a two layer deep learning model.

A dynamic one layer auto-encoder for a financial time series (Y_t) can, for example, be written as a coupled system of the form

$$Y_t = W_x X_t + W_y Y_{t-1} \text{ and } \begin{pmatrix} X_t \\ Y_{t-1} \end{pmatrix} = W Y_t \,.$$

We then need to learn the weight matrices W_x and W_y. Here, the state equation encodes and the matrix W decodes the Y_t vector into its history Y_{t-1} and the current state X_t.

The auto-encoder demonstrates nicely that in deep learning we do not have to model the variance-covariance matrix explicitly, as our model is already directly in predictive form. (Given an estimated non-linear combination of deep learners, there is an implicit variance-covariance matrix, but that is not the driver of the method.)

5 Application: Smart Indexing for the Biotechnology IBB Index

We consider weekly returns data for the component stocks of the biotechnology IBB index for the period January 2012 to April 2016. We train our learner without knowledge of the component weights. Our goal is to find a selection of investments for which good out-of-sample tracking properties of our objective can be found.

5.1 Four Step Algorithm

Assume that the available market data has been separated into two (or more for an iterative process) disjoint sets for training and validation, respectively, denoted by X and $\hat{X}$.

Our goal is to provide a self-contained procedure that illustrates the trade-offs involved in constructing portfolios to achieve a given goal, e.g., to beat a given index by a pre-specified level. The projected real-time success of such a goal will depend crucially on the market structure implied by our historical returns. (While not explicitly investigated here, there is also the possibility of including further conditioning variables during our training phase. These might include accounting information or further returns data in the form of derivative prices or volatilities in the market.)

Our four step deep learning algorithm proceeds via auto-encoding, calibrating, validating, and verifying. This data driven and model independent approach provides a new paradigm for prediction and can be summarized as follows. (See also Hutchinson et al., 1994. To contextualize within classic statistical methods, e.g., see Wold, 1956, or Hastie et al., 2009.)

I. Auto-encoding

Find the *market-map*, denoted by $F_W^m(X)$, that solves the regularization problem

$$\min_W \|X - F_W^m(X)\|_2^2 \quad \text{subject to} \quad \|W\| \leq L^m. \tag{3}$$

For appropriately chosen F_W^m, this auto-encodes X with itself and creates a more information-efficient representation of X (in a form of *pre-processing*).

II. Calibrating

For a desired result (or target) Y, find the *portfolio-map*, denoted by $F^p_W(X)$, that solves the regularization problem

$$\min_W \|Y - F^p_W(X)\|_2^2 \quad \text{subject to} \quad \|W\| \leq L^p. \tag{4}$$

This creates a (non-linear) portfolio from X for the approximation of objective Y.

III. Validating

Find L^m and L^p to suitably balance the trade-off between the two errors

$$\epsilon_m = \|\hat{X} - F^m_{W^*_m}(\hat{X})\|_2^2 \quad \text{and} \quad \epsilon_p = \|\hat{Y} - F^p_{W^*_p}(\hat{X})\|_2^2,$$

where W^*_m and W^*_p are the solutions to (3) and (4), respectively.

IV. Verifying

Choose *market-map* F^m and *portfolio-map* F^p such that validation (step 3) is satisfactory.

A central observation to the application of our four step procedure in a finance setting is that univariate activation functions can frequently be interpreted as compositions of financial put and call options on linear combinations of the input assets. As such, the deep feature abstractions implicit in a deep learning routine become *deep portfolios*, and are investible – which gives rise to a *deep portfolio theory*. Put differently, deep portfolio theory relies on deep features, lower (or hidden) layer abstractions which, through training, correspond to the independent variable.

The question is how to use training data to construct the deep portfolios. The theoretical flexibility to approximate virtually any non-linear payout function puts regularization in training and validation at the center of deep portfolio theory. In our four step procedure, portfolio optimization and inefficiency detection become an almost entirely data driven (and therefore model-free) tasks, contrasting with classic portfolio theory.

When plotting the goal of interest as a function of the amount of regularization, we refer to this as the efficient *deep frontier*, which serves as a metric during the verification step.

5.2 Smart Indexing the IBB Index

For the four phases of our deep portfolio process (*auto-encode, calibrate, validate*, and *verify*), we conduct auto-encoding and calibration on the period January 2012 to December 2013, and validation and verification on the period January 2014 to April 2016. For the auto-encoder as well as the deep learning routine, we use one hidden layer with five neurons.

After *auto-encoding* the universe of stocks, we consider the 2-norm difference between every stock and its auto-encoded version and rank the stocks by this measure of degree of *communal information*. (In reproducing the universe of stocks from a bottleneck network structure, the auto-encoder reduces the total information to an information subset which is applicable to a large number of stocks. Therefore, proximity of a stock to its auto-encoded version provides a measure for the

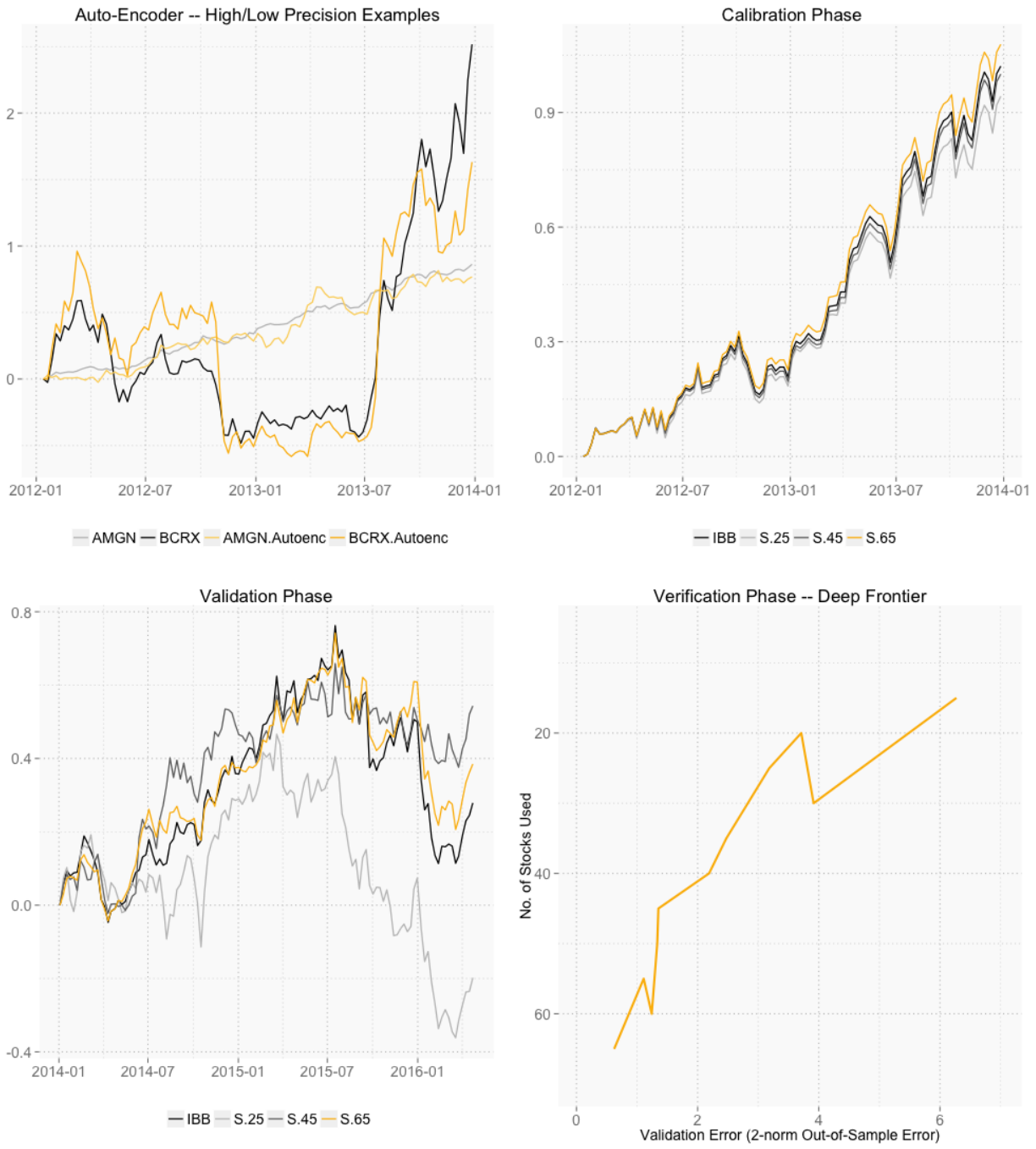

Figure 1: *We see the four phases of a deep portfolio process: Auto-encode, Calibrate, Validate, Verify. For the auto-encoder as well as the deep learning routine, we use one hidden layer with five neurons. We use ReLU activation functions. We have a list of component stocks but no weights. We want to select a subset of stocks and infer weights to track the IBB index. S25, S45, etc. denotes number of stocks used. After ranking the stocks in auto-encoding, we are increasing the number of stocks by using the 10 most communal stocks plus x-number of most non-communal stocks (as we do not want to add unnecessary communal information); e.g., 25 stocks means 10 plus 15 (where x=15). We use weekly returns and 4-fold cross validation in training. We calibrate on the period Jan-2012 to Dec-2013, and then validate on the period Jan-2014 to Apr-2016. The deep frontier (bottom right) shows the trade-off between the number of stocks used and the validation error.*

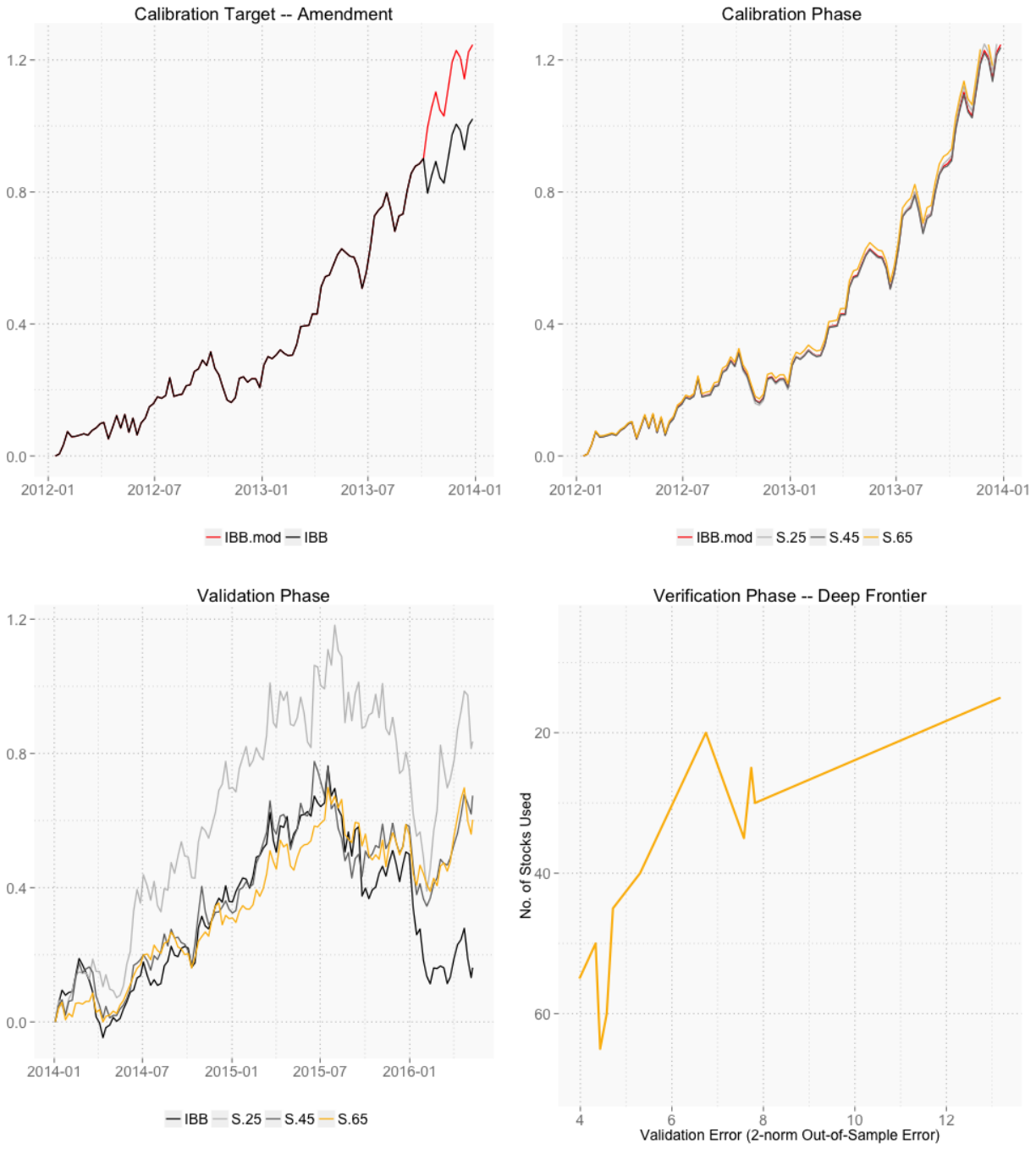

Figure 2: *We proceed exactly as in Figure 1, but we alter the target index in the calibration phase by replacing all returns* $< -5\%$ *by exactly* 5%*, which aims to create an index tracker with anti-correlation in periods of large drawdowns. On the top left, we see the altered calibration target. During the validation phase (bottom left) we notice that our tracking portfolio achieves the desired returns in periods of drawdowns, while the deep frontier (which is calculated with respect to the modified target on the validation set, bottom right) shows that the expected deviation from the target increases somewhat throughout compared to Figure 1 (as would be expected).*

similarity of a stock with the stock universe.) As there is no benefit in having multiple stocks contributing the same information, we increasing the number of stocks in our deep portfolio by using the 10 most communal stocks plus x-number of most non-communal stocks (as we do not want to add unnecessary communal information); e.g., 25 stocks means 10 plus 15 (where $X = 15$). In the top-left chart in Figure 1, we see the stocks AMGN and BCRX with their auto-encoded versions as the two stocks with the highest and lowest communal information, respectively.

In the *calibration* phase, we use rectified linear units (ReLU) and 4-fold cross-validation. In the top-right chart in Figure 1, we see training results for deep portfolios with 25, 45, and 65 stocks, respectively.

In the bottom-left chart of Figure 1, we see *validation* (i.e. *out-of-sample* application) results for the different deep portfolios. In the bottom-right chart in Figure 1, we see the *efficient deep frontier* of the considered example, which plots the number of stocks used in the deep portfolio against the achieved validation accuracy.

Model selection (i.e. *verification*) is conducted through comparison of efficient deep frontiers.

While the efficient deep frontier still requires us to choose (similarly to classic portfolio theory) between two desirables, namely index tracking with few stocks as well as a low validation error, these decisions are now purely based on out-of-sample performance, making deep portfolio theory a strictly *data driven approach*.

5.3 Outperforming the IBB Index

The *1%-problem* seeks to find the best strategy to outperform a given benchmark by 1% per year. In our theory of deep portfolios, this is achieved by uncovering a performance improving *deep feature* which can be trained and validated successfully. Crucially, thanks to the Kolmogorov-Arnold theorem (see Section 2), hierarchical layers of univariate non-linear payouts can be used to scan for such features in virtually any shape and form.

For the current example (beating the IBB index), we have amended the target data during the calibration phase by replacing all returns smaller than -5% by exactly 5%, which aims to create an index tracker with anti-correlation in periods of large draw-downs. We see the amended target as the red curve in the top-left chart in Figure 2, and the training success on the top-right.

In the bottom-left chart in Figure 2, we see how the *learned* deep portfolio achieves outperformance (in times of draw-downs) during validation.

The efficient deep frontier in the bottom-right chart in Figure 2 is drawn with regard to the amended target during the validation period. Due to the more ambitious target, the validation error is larger throughout now, but, as before, the verification suggests that, for the current model, a deep portfolio of at least forty stocks should be employed for reliable *prediction*.

6 Conclusion

Deep learning presents a general framework for using large data sets to optimize predictive performance. As such, deep learning frameworks are well-suited to many problems – both practical and theoretical – in finance. This paper introduces deep learning hierarchical decision models for problems in financial prediction and classification. Deep learning has the potential to improve – sometimes dramatically – on predictive performance in conventional applications. Our example on smart indexing in Section 5 presents just one way to implement deep learning models in finance. Sirignano (2016) provides an application to limit order books. Many other applications remain for development.

References

Cook, R. D. (2007). *Fisher Lecture: Dimension Reduction in Regression*, **Statistical Science**, pp. 1-26.

Dean, J., Corrado, G., Monga, R., et al. (2012). *Large Scale Distributed Deep Networks*, **Advances in Neural Information Processing Systems**, pp. 1223-1231.

Diaconis, P. and Shahshahani, M.(1984). *On Non-linear Functions of Linear Combinations*, **SIAM Journal on Scientific and Statistical Computing**, Vol. 5(1), pp. 175-191.

Gallant, A. R. and White, H. (1988). *There exists a Neural Network that does not make Avoidable Mistakes*, **IEEE International Conference on Neural Networks**, Vol. 1, pp. 657-664.

Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning*, Vol 2.

Hinton, G. E. and Salakhutdinov, R. R. (2006). *Reducing the Dimensionality of Data with Neural Networks*, **Science**, Vol. 313(5786), pp. 504-507.

Hornik, K., Stinchcombe, M., and White, H. (1989). *Multilayer Feedforward Networks are Universal Approximators*, **Neural networks**, Vol. 2(5), pp. 359-366.

Hutchinson, J. M., Lo, A. W., and Poggio, T. (1994). *A Nonparametric Approach to Pricing and Hedging Derivative Securities via Learning Networks*, **Journal of Finance**, Vol. 48(3), pp. 851-889.

Kolmogorov, A. (1957). *The Representation of Continuous Functions of Many Variables by Superposition of Continuous Functions of one Variable and Addition*, **Dokl. Akad. Nauk SSSR**, Vol. 114, pp. 953-956.

Lake, B. M., Salakhutdinov, R., and Tenenbaum, J. B. (2015). *Human-Level Concept Learning through Probabilistic Program Induction.* **Science**, Vol. 3560, pp. 1332-1338.

LeCun, Y.A., L. Bottou, G. B. Orr, and K.-R. Muller (2012). *Efficient Backprop*, **Neural Networks: Tricks of the Trade**, pp. 9-48.

Lorentz, G. G. (1976). *The 13th Problem of Hilbert*, **Proceedings of Symposia in Pure Math-**

ematics, American Mathematical Society, Vol. 28, pp. 419-430.

Poggio, T. and F. Girosi (1990). *Networks for Approximation and Learning*, **Proceedings of the IEEE**, Vol. 78(9), pp. 1481-1497.

Polson, N. G., Scott, J. G., and Willard, B. T. (2015). *Proximal Algorithms in Statistics and Machine Learning*, **Statistical Science**, Vol. 30, pp. 559-581.

Ripley, B. D. (1996). *Pattern Recognition and Neural Networks*. **Cambridge University Press**.

Sirignano, J. (2016). *Deep Learning for Limit Order Books*. **arXiv** 1601.01987v7.

Srivastava et al. (2014). *Dropout: a Simple Way to Prevent Neural Networks from Overfitting*, **Journal of Machine Learning Research**, Vol. 15, pp. 1929-1958.

Stein, C. (1981). *Estimation of the Mean of a Multivariate Normal Distribution*. **Journal of the American Statistical Association**, Vol. 97, pp. 210-221.

Wold, H. (1956). *Causal Inference from Observational Data: a Review of End and Means*, **Journal of the Royal Statistical Society**, Series A (General), pp. 28-61.