ONLINE APPENDIX
Can Deep Learning Predict Risky Retail Investors? A Case Study in Financial Risk Behavior Forecasting

A. Kim[a], Y. Yang[c], S. Lessmann[a], T. Ma[b], M.-C. Sung[b,*], J.E.V. Johnson[b]

[a]*School of Business and Economics, Humboldt-University of Berlin*
[b]*Southampton Business School, University of Southampton*
[c]*Department of Computer Science, University College London*

Abstract

The online appendix complements the paper through i) further elaborating on the training of the trader classification DNN, ii) documenting candidate hyper-parameter settings of the DNN and the ML benchmark classifiers and how we have tuned these in a model selection, and iii) providing additional empirical results. These results include empirical findings from pre-tests based on a static split sample design. We also report the distributions of the performance indicators that we have used in the comparison of the proposed DNN to benchmark ML classifiers. Finally, we provide additional results of a comparison of the DNN to alternative DL models.

Keywords: risk management, retail finance, forecasting, deep learning

1. Training the Proposed DNN

The DNN employed in the paper integrates multiple DL concepts. In summary, the DNN is first pre-trained via SdA, with Xavier's initialisation for weights and ReLU unit as the activation function, in a greedy layer-wiser manner. This is the unsupervised pre-tuning stage after which we fine-tune the DNN as a whole in a supervised way, with each hidden layer followed by batch normalisation and dropout. In the following, we detail the optimisation of the cost function, , which we did not detail in the main body of the paper for brevity; namely, the training of the parameters of the DNN in both the pre-training and fine-tuning stage as well as other DL concepts such as Xavier's initialization, ReLU, and batch normalization.

The parameters to train in the pre-training stage are the weight matrix and bias in each dA (both the encoder and the decoder), and the parameters to train in the supervised fine-tuning stage are the weight matrix and bias in each encoder of SdA and in the *softmax* regression. The rest of the parameters (e.g., the number of hidden layers in SdA, the number of hidden neurons in each hidden layer), are hyper-parameters that need adjusting on top of the training process. We elaborate on the treatment of such hyper-parameters below in Section 2.

*Corresponding author
Email addresses: alisa.k@protonmail.com (A. Kim), yaodong.yang@outlook.com (Y. Yang), stefan.lessmann@hu-berlin.de (S. Lessmann), tiejun.ma@soton.ac.uk (T. Ma), M.SUNG@soton.ac.uk (M.-C. Sung), J.E.Johnson@soton.ac.uk (J.E.V. Johnson)

1.1. Xavier's initialization

The solution to a non-convex optimization problem depends on the initial values of the weight parameters W. By default, SdA initialize weights randomly. If network weights are initialized too small (large), the signal shrinks (expands) as it passes through each layer until it becomes too tiny (massive) to be useful. Xavier's initialization [1] guarantees that weights are sensibly initialized by ensuring that the variance of the input and output signals passed through the network remain the same.

$$Var(W_i) = \frac{2}{n_{in} + n_{out}} \tag{1}$$

where W_i is the weight matrix in layer i and n_{in} and n_{out} are the number of neurons feeding in and out.

1.2. ReLU

Using non-linear coding functions $h(\cdot)$ and $g(\cdot)$ enables a dA to discover intricate non-linear structures from the input data. It has become common practice to replace the *sigomid* function with a ReLu in the encoding part. The activation functions in dA are then:

$$h(x) = \quad \text{ReLU}(x) = \quad \max(0, x) \tag{2}$$

ReLU outperforms other non-linear transformation functions on a majority of ML tasks [2]. Setting half of the outputs to zero, it creates robust and sparse representations, which is beneficial for learning algorithms [3]. In addition, ReLU does not require any exponential computation, which substantially accelerates learning. Moreover, its derivative is a step function that provides the network with more non-linearities. These become paramount if stacking a multitude of dAs to build a DNN [4]. For example, ReLU does not suffer from the gradient explosion/vanishing problem [5].

1.3. Batch normalization

During DNN training, the distribution of each layer's inputs changes with updates of the parameters of the preceding layers. With greater depth, small changes to network parameters are amplified; thus, the layers need to keep adapting to the new distribution. Enforcing lower learning rates, this problem decelerates the training process. A batch normalization layer fixes the means and variances of layer outputs according to 3 [6]. It whitens each feature independently after it passes through an activation function in the hidden layer. Moreover, instead of using the whole training sample, the mean and variance in the whitening process are estimated in a batch-wise manner so that the training of layer parameters $(\boldsymbol{\gamma}, \boldsymbol{\beta})$ can be integrated into the original back-propagation algorithm

$$\hat{\mathbf{x}} = \boldsymbol{\gamma} \cdot \frac{\mathbf{x} - E[\mathbf{x}]}{\sqrt{\text{Var}[\mathbf{x}] + \epsilon}} + \boldsymbol{\beta} \tag{3}$$

1.4. Stochastic gradient descent

SGD has been shown to be an effective tool to train the DNN. It is an extension of ordinary gradient descent. In ordinary gradient descent, the model parameters θ are repeatedly updated by taking small steps downward on an error surface that is defined by an objective function $E(\theta)$; here it is the cost function $L_{(\theta,\mathbf{x})}$ as Equation (3) or (6) in the main paper. At each iteration e, the step size is set by the learning rate ε, and the direction of each step equals the back-propagated gradient of the objective function over the N entire training set:

$$\nabla E(\boldsymbol{\theta},\mathbf{x}) = \frac{\partial L(\boldsymbol{\theta},\mathbf{x})}{\partial \boldsymbol{\theta}} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial L(\boldsymbol{\theta},x_i)}{\partial \boldsymbol{\theta}} \tag{4}$$

$$\begin{aligned}\boldsymbol{\theta}_e &= \boldsymbol{\theta}_{e-1} - \varepsilon\nabla E(\boldsymbol{\theta}_{e-1},\mathbf{x})\\ &= \boldsymbol{\theta}_{e-1} - \frac{\varepsilon}{N}\sum_{i=1}^{N}\frac{\partial L(\boldsymbol{\theta}_{e-1},x_i)}{\partial \boldsymbol{\theta}_{e-1}}\end{aligned} \tag{5}$$

SGD works identically to ordinary gradient descent, except that in each iteration it only uses a "batch" (a subset m of N) of training samples in computing the gradient. The training samples are divided into multiple batches in advance. SGD iterates through different batches and updates the parameters until the value of the cost function stops decreasing (hitting the optimum).

$$\nabla E(\boldsymbol{\theta},\mathbf{x}) = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial L(\boldsymbol{\theta},x_i)}{\partial \boldsymbol{\theta}} \tag{6}$$

SGD speeds up the convergence because in every iteration when updating the values of the parameters, it is not necessary to run through the complete training set in order to update the parameters. It is "stochastic" because using batch approximations introduces noise in estimating the true gradient: the gradient over all training samples. Although such estimation is biased, it leads the DNN to skip over poor local minima that traditional GD would fall into. Meanwhile, it reduces variance of the gradients [7], which helps prevent the DNN from overfitting [8]. Most importantly, SGD makes better use of the memory allocation in computers since the training data in large scale ML task are usually too large to be fitted into the local memory.

1.5. Momentum

Momentum [9] has become a common trick in achieving the state-of-the-art performance. If the objective function has the pattern of a long shallow ravine leading to the optimum with steep walls on the sides (a deep "U" shape with optimum at the bottom), the SGD tends to oscillate across the optimum since the gradient will point down to steep sides rather than along the way direct towards the optimum. Deep architectures are shown to have similar patterns near the local minima [10], therefore, ordinary SGD can lead to slow convergence; particularly after the initial steep gains. The momentum technique is one way to pull the objective function along the shallow ravine. It speeds up the convergence, and reduces the risk of oscillating by incorporating the gradient information from previous steps. Mathematically, the momentum term $\mu \in (0, 1]$ determines to what extent previous gradients are combined into the current update. As a rule of thumb, μ is set to 0.5 in first epochs. After the training stabilises, it is increased to 0.9 or higher in later iterations. In epoch e, the model parameters θ are updated as Equation (7).

$$\begin{aligned}\boldsymbol{\theta}_e &= \boldsymbol{\theta}_{e-1} - \Delta\boldsymbol{\theta}_e\\ \Delta\boldsymbol{\theta}_e &= \mu_e\Delta\boldsymbol{\theta}_{e-1} + (1-\mu_e)\varepsilon_e\nabla E(\boldsymbol{\theta}_{e-1})\end{aligned} \tag{7}$$

1.6. Decaying learning rate

SGD is sensitive to the learning rate since it indicates how big a step should be taken in each update. The intuition in choosing the learning rate is that we should decrease it as the number of updates increases, otherwise the training process will just oscillate near the local minima. A naive implementation is that we decay the learning rate by a certain percent after each epoch (an epoch is one run that SGD iterates over all the batches of training data). There are other advanced methods, such as Adagrad [11] and Adadelta [12], that can dynamically set a learning rate.

1.7. Early stopping

Improving the DNN's fitness to the training data often comes at the cost of increased generalisation error; i.e the overfitting problem. In order to prevent overfitting, we stop the SGD procedures earlier, before the cost function converges to the true best minimum. Early stopping technique [13] offers guidance as to how many updates are allowed before the learner starts to overfit. Such guidance is based on measuring the model's real-time performance on a pre-allocated validation set. If the performance on the validation set stops improving for a long time and it exceeds the limit of patience (a self-defined parameter), then the training process will be stopped because it has probably met the local minima.

1.8. Implementation Details

We have implemented the simple benchmark classifiers (i.e., logistic regression, Naive bayes, decision tree) with the help of the *scikit-learn* library [14] in Python. The *Scikit-learn* library offers the option of weighted class samples. To take the unbalanced class issue into account, we tried different weight values, ranging from 1 to 15, and selected the one with the best performance for each algorithm. Other hyper-parameters in the library API were set at default, other than those specifically mentioned in Section 2 in the online Appendix. For more advanced ML algorithms (i.e., SVM, ensemble methods), we have employed a *big data* framework: Hadoop + Spark. ML algorithms with iterative calculations often prohibit researchers from performing large scale data analysis. For example, in the work of comparing ML algorithms in the problem of mortgage default prediction, even with $300,000$ training samples, SVM is regarded as computationally infeasible due to its $\mathrm{O}(N^3)$ complexity [15]. By using a special data abstraction called Resilient Distributed Dataset (RDD) and caching the RDD into memory, Spark offsets the weakness of low efficiency on running iterative jobs for a traditional Hadoop framework [16]. This makes machine learning on big data possible. We deployed the Hadoop + Spark architecture on Amazon EC2. Nineteen $m1.xlarge$ salves each with 4 cores and $12.6gb$ memory were used.

1.9. GPU Implementation

DNN has a massively parallel structure. Training DNNs heavily depends on matrix calculations which can be computed simultaneously. This makes the training task perfectly suitable for speeding up by graphics processing units (GPUs). A GPU has thousands of cores, and can thus support large scale of parallelisations. In the task of training DNNs, it can offer 20 times faster speeds compared to the CPUs [17]. With the aid of GPU, we were able to train huge DNNs (e.g. millions of parameters) in a timely manner.

Algorithm 1 Pseudo code for training DNN

Input: Training data set $(\mathbf{X}_{train}, \mathbf{Y}_{train})$, N samples with P features; Validation data set $(\mathbf{X}_{valid}, \mathbf{Y}_{valid})$, N' samples with P features. $\mathbf{X}_{train}$ and $\mathbf{X}_{valid}$ are normalised before input.

Input: Number of hidden layers: I, the i-th hidden layer with h^i neurons; Corruption rate in the i-th layer of dA: q_i; Activation functions in encoders or decoders: $h(\cdot), g(\cdot)$; Cost function in layer-wise pre-training or fine-tuning: $L_p(\cdot), L_f(\cdot)$

Input: Learning rate of pre-training or fine-tuning in the e-th epoch: $\varepsilon_{p\_e}, \varepsilon_{f\_e}$; Momentum in the e-th epoch: u_e; Batch size in SGD: B; Maximum number of epochs in pre-training or fine-tuning stage: N, M; Dropout rate in the i-th hidden layer: p_i

Output: DNN for later inference, e.g., making predictions on out-of-sample dataset

```
 1: **Xavier's Initialisation**: $i$-th layer encoder / decoder bias and weights $[\mathbf{b}_i, \tilde{\mathbf{b}_i}] = 0$,
    $[\mathbf{W}_i, \tilde{\mathbf{W}}_i] \sim \text{Gaussian}(0, \frac{2}{h^{i-1} + h^i})$
 2: // Layer-wise unsupervised pre-training via denoising autoencoder:
 3: $\mathbf{x_0} = \mathbf{X}_{train},\ \ e = 0$        // setting the input and the epoch counter
 4: for $i \in (1, 2..., I)$ do
 5:   // encoder's outputs from previous layer is used as the input for the subsquent layer
 6:   $\mathbf{x_i} = h(\mathbf{W}_{i-1} \cdot \mathbf{x}_{i-1} + \mathbf{b}_{i-1})$
 7:   while $e \leq N$ do
 8:     for $j \in (1, 2..., \frac{N}{B})$ do
 9:       // Corrupting the $j$-th input batch of data $\mathbf{x}_i^j$ by randomly knocking out samples
10:       $\hat{\mathbf{x}}_i^j \leftarrow \mathbf{x}_i^j * \mathbf{Binomial}\,(n = B, p = q_i)$
11:       // Computing the reconstruction of $\tilde{\mathbf{x}}_i^j$ through the encoder and decoder:
          $\mathbf{z}_i^j = g(\tilde{\mathbf{W}}_i^e \cdot h(\mathbf{W}_i^e \cdot \hat{\mathbf{x}}_i^j + \boldsymbol{b}_i^e) + \tilde{\boldsymbol{b}_i^e})$
12:       // Computing the reconstruction error:
          $L_p(\hat{\mathbf{x}}_\mathbf{i}^\mathbf{j}, \mathbf{z}_\mathbf{i}^\mathbf{j} \mid \boldsymbol{\Theta}) = L_p(\hat{\mathbf{x}}_\mathbf{i}^\mathbf{j}, \mathbf{z}_\mathbf{i}^\mathbf{j} \mid \mathbf{W}_i^e, \mathbf{b}_i^e, \tilde{\mathbf{W}}_i^e, \tilde{\boldsymbol{b}_i^e})$
13:       // Computing the average gradient among a batch and update the parameters:
          $\mathbf{W}_i^{e+1} \leftarrow \mathbf{W}_i^e - \frac{\varepsilon_{p\_e}}{B} \sum_{k=1}^{B} \frac{\partial L_p(\hat{x}_i^{j\_k}, z_i^{j\_k} | \boldsymbol{\Theta})}{\partial \mathbf{W}_i^e}, \mathbf{b}_i^{e+1} \leftarrow \mathbf{b}_i^e - \frac{\varepsilon_{p\_e}}{B} \sum_{k=1}^{B} \frac{\partial L_p(\hat{x}_i^{j\_k}, z_i^{j\_k} | \boldsymbol{\Theta})}{\partial \mathbf{b}_i^e}$
          $\tilde{\mathbf{W}}_i^{e+1} \leftarrow \tilde{\mathbf{W}}_i^e - \frac{\varepsilon_{p\_e}}{B} \sum_{k=1}^{B} \frac{\partial L_p(\hat{x}_i^{j\_k}, z_i^{j\_k} | \boldsymbol{\Theta})}{\partial \tilde{\mathbf{W}}_i^e}, \tilde{\mathbf{b}}_i^{e+1} \leftarrow \tilde{\mathbf{b}}_i^e - \frac{\varepsilon_{p\_e}}{B} \sum_{k=1}^{B} \frac{\partial L_p(\hat{x}_i^{j\_k}, z_i^{j\_k} | \boldsymbol{\Theta})}{\partial \tilde{\mathbf{b}}_i^e}$
14:     end for
15:     if $L_p(\mathbf{X}_{valid} \mid \boldsymbol{\Theta})$ meets the early stopping condition then
16:       Save the weights and bias of the encoder in each layer
17:       break
18:     end if
19:     e += 1
20:   end while
21: end for
```

```
22: // Supervised fine-tunning the whole network:
23: Initialise parameters in the batch normalisation layer: $\boldsymbol{\gamma}_i \sim \text{Uniform}(-h^i, h^i), \boldsymbol{\beta}_i = 0$
24: **Xavier's Initialisation**: softmax layer weight $\hat{\mathbf{W}} \sim \text{Gaussian}(0, \frac{2}{h^i + 2})$, bias $\hat{\mathbf{b}} = 0$
25: $\mathbf{x_0} = \mathbf{X}_{train}, \ \ e = 0$        // resetting the input and the epoch counter
26: **while** $e \leq M$ **do**
27:     **for** $j \in (1, 2..., \frac{N}{B})$ **do**
28:         // Feed forward to compute the outputs for each batch of training data:
29:         **for** $i \in (1, 2..., I)$ **do**
30:             $\mathbf{x}_i^j = h(\mathbf{W}_i^e \cdot \mathbf{x}_{i-1}^j + \mathbf{b}_i^e)$        // inherit the weights and bias from pretrained encoder
31:             $\hat{\mathbf{x}}_i^j = \boldsymbol{\gamma}_i^e \cdot \frac{\mathbf{x}_i^j - E[\mathbf{x}_i^j]}{\sqrt{\text{Var}[\mathbf{x}_i^j] + \epsilon}} + \boldsymbol{\beta}_i^e$        // batch normalisation layer
32:             // Drop out the neurons with their corresponding weights and outputs:
33:             $\hat{\mathbf{x}}_i^j \leftarrow \hat{\mathbf{x}}_i^j * \textbf{Binomial}(n = h^i, p = p_i)$
34:         **end for**
35:         $\mathbf{Y}_{predict}^j = \text{softmax}(\hat{\mathbf{W}}^e \cdot \hat{\mathbf{x}}_I^j + \hat{\mathbf{b}}^e)$        // softmax layer outputs the predictions
36:         // Computing the cost function:
            $L_f(\mathbf{Y}_{predict}^j, \mathbf{Y}_{train}^j \mid \boldsymbol{\Theta}) = L_f(\mathbf{Y}_{predict}^j, \mathbf{Y}_{train}^j \mid \mathbf{W}_{1\sim I}^e, \mathbf{b}_{1\sim I}^e, \hat{\mathbf{W}}^e, \hat{\boldsymbol{b}}^e, \boldsymbol{\gamma}_{1\sim I}^e, \boldsymbol{\beta}_{1\sim I}^e)$
37:         // Computing the gradient and update the parameters of the softmax layer
38:         $\boldsymbol{\theta} : \{\hat{\mathbf{W}}, \hat{\mathbf{b}}\}$        // parameters to update
39:         $\Delta\boldsymbol{\theta}^{e+1} = \mu_e \Delta\boldsymbol{\theta}^e + (1 - \mu_e)\frac{\varepsilon_{f\_e}}{B} \sum_{k=1}^{B} \frac{\partial L_f(\mathbf{Y}_{predict}^{j\_k}, \mathbf{Y}_{train}^{j\_k} | \boldsymbol{\Theta})}{\partial \boldsymbol{\theta}^e}$        // with momentum
            $\boldsymbol{\theta}^{e+1} \leftarrow \boldsymbol{\theta}^e - \Delta\boldsymbol{\theta}^{e+1}$
40:         // Propagating back gradients and update intermediate layers
41:         **for** $i' \in (l, l-1, ..., 1)$ **do**
42:             $\boldsymbol{\theta} : \{\mathbf{W}_{i'}, \mathbf{b}_{i'}, \boldsymbol{\gamma}_{i'}, \boldsymbol{\beta}_{i'}\}$        // parameters of hidden and batch normalisation layers
43:             // Applying chain rules
44:             $\Delta\boldsymbol{\theta}^{e+1} = \mu_e \Delta\boldsymbol{\theta}^e + (1 - \mu_e)\frac{\varepsilon_{f\_e}}{B} \sum_{k=1}^{B} \left\{ \frac{\partial L_f(\mathbf{Y}_{predict}^{j\_k}, \mathbf{Y}_{train}^{j\_k} | \boldsymbol{\Theta})}{\partial \hat{\mathbf{x}}_l^j} \cdot \prod_{i^*=l}^{i'+1} \left\{ \frac{\partial \hat{\mathbf{x}}_{i^*}^j}{\hat{\mathbf{x}}_{i^*-1}^j} \right\} \cdot \frac{\partial \hat{\mathbf{x}}_{i'}^j}{\partial \boldsymbol{\theta}^e} \right\} \boldsymbol{\theta}^{e+1} \leftarrow$
                $\boldsymbol{\theta}^e - \Delta\boldsymbol{\theta}^{e+1}$
45:         **end for**
46:     **end for**
47:     **if** $L_p(\mathbf{Y}_{predict}^j, \mathbf{Y}_{train}^j \mid \boldsymbol{\Theta})$ meets the early stopping condition **then**
48:         Save the DNN
49:         **break**
50:     **end if**
51:     e += 1
52: **end while**
```

2. Hyper-Parameter Tuning for the DNN and Benchmark ML Classifiers

The (predictive) performance of a learning algorithm depends on the setting of algorithmic hyper-parameters. Tuning hyper-parameters in a model selection stage is important to ensure that an algorithm performs well on a given data set [18]. We tune the proposed DNN and ML benchmark classifiers in such a way that we reserve a fraction of 20% of the training set as a validation data to assess candidate models with different hyper-parameter settings. We then use the model with the best hyper-parameter configuration in terms of validation set performance to generate risk predictions for the trades in the test set. Given that we employ n-fold cross-validation, the model training and evaluation occurs n times. In theory, this suggests that the selection of suitable hyper-parameters should also be undertaken n times; once for each loop of cross-validation. Given the computational effort associated with training advanced learning algorithms on a large data set and the number of alternative hyper-parameter settings, repeating model selection n times is computationally infeasible. Therefore, we perform model selection only once in the first iteration of cross-validation. In subsequent iterations, we retrain the learning algorithms using the hyper-parameter specification identified as most suitable in the first cross-validation iteration.

We acknowledge that our model selection approach suffers the limitation that it uses only a relatively small amount of data to tune hyper-parameters. More specifically, when setting $n = 10$ in cross-validation, training sets comprise roughly 90% of the available data out of which 20% are used as validation set. Therefore, the single validation partition on which we assess the predictive performance of candidate hyper-parameter settings is roughly 18% of the full data set. While computational considerations render model selection in every round of cross-validation infeasible, we suggest that our approach is also suitable. Our motivation for this view is twofold. First, 18% of the full data set are still a sizeable amount of data in absolute terms, as we work with a large data set including about 30 million trades. Second, the DNN classifier exhibits the largest number of different hyper-parameters and hyper-parameter candidate settings in the comparison. DNNs are also considered sensitive with respect to hyper-parameter choices, which suggest that model selection is particularly important for DNNs. Consequently, reducing the amount of hyper-parameter tuning in our approach compared to a full model selection in every cross-validation iteration provides a conservative evaluation of the ability of the DNN. If the available resources facilitates a more comprehensive tuning of hyper-parameters, it is plausible to expect the DNN to benefit the most from such additional tuning.

The range of candidate settings that we consider for each learning algorithm is based on previous literature, while accounting for both the large size of the data set and computational feasibility. More specifically, we draw inspiration from previous classifier comparisons [18, 19, 15, 20] to identify candidate settings for the ML benchmarks. For the DNN, we follow the recommendations of [21] and consider the candidate hyper-parameter settings he proposes. We also follow the advice of [21] to not tune DNN hyper-parameters using grid-search, which would involve a full-enumerative search across all combinations of hyper-parameter candidate settings that is computationally intractable, but to use random search. Other than using random search, the tuning process for the DNN is the same as that for the ML benchmarks.

Table 1 and Table 2 report the candidate hyper-parameter settings that we consider during the tuning of the DNN and the ML benchmarks, respectively.

Table 1: Candidate Hyper-Parameter Settings for the Proposed DNN

Attribution	Hyper-parameter	Range Selected
DNN Topology	Number of hidden layers (dAs)	[2, 3, 4, 5, 6]
	Number of hidden units in each hidden layer[a]	[32, 64, 128, 256, 512, 1024, 2048]
	Weight decay regularizer λ in dA	[10^{-2}, 10^{-3}, 10^{-4}, 10^{-2}, 10^{-3}, 10^{-5}]
	Corruption rate in each hidden layer[b]	[0.2, 0.3, 0.4, 0.5]
SGD Training	Learning rate in pre-training	[1, 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}]
	Learning rate in fine-tuning	[1.5, 1, 0.5, 0.1, 0.05, 0.01]
	Learning rate decay[c]	[0.99, 0.995, 0.999]
	Number of samples in minibatch	[10, 20, 30, 50, 100, 150, 200]
	Momentum interval[d]	[200, 500, 800]
	Momentum start[d]	0.5
	Momentum end[d]	[0.9, 0.99]
	Number of epochs in pre-training	[30, 50, 70]
	Maximum number of epochs in fine-tuning[e]	[500, 1000, 1500]
Dropout	Dropout rate in each layer[f]	0.5

Notes: **a**) The number of neurons in the middle layers is selected larger than the the number of hidden units in the first and top layers, e.g., 4 hidden layers: [64, 256, 512, 32]. **b**) The corruption rate is set in an increasing way, e.g., 4 hidden layers: [0.2, 0.3, 0.4, 0.5]. **c**) In the pre-training stage, the learning rate is fixed. In the fine-tuning stage, the learning rate decays by a given percent in each epoch. **d**) For epoch $e \in [0, \textit{momentum interval}]$, the momentum μ_e increases linearly from *momentum start* to *momentum end*. After that, it stays at *momentum end*. **e**) In the fine-tuning stage, we also use early stopping. Be aware that the training process can stop before the current epoch reaches the maximum number. **f**) The dropout rate is the same for all dropout layers.

Table 2: Candidate Hyper-Parameter Settings for ML Benchmark Classifiers

Algorithm	**Hyper-Parameter**	**Candidate Settings**
Naive Bayes	n.A.	
Logistic Regression	form of regularization	none, L2, L1, forward selection
	regularizer	$\{ 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1, 10^{1}, 10^{2}, 10^{3} \}$
Artificial Neural Network	number of hidden units	2, 8, 31, 64, 128
Support Vector Machine	kernel function	linear
	regularizer	$\{ 10^{-4},10^{-3},10^{-2},10^{-1},1,10^{1},10^{2},10^{3}\}$
C5.0 Tree	splitting criterion	entropy , gini-impurity
	max. depth	2, 4, 6, 8, 12, 16,20
Random Forest	number of trees	32, 64, 128, 256, 512, 1024, 2048
	max. depth	1, 2, 4, 8, 20
	random subspace	2, 5, 10, 15, 31
Adaptive boosting	number of trees	32, 64, 128, 256, 512, 1024, 2048
	max. depth	1, 2, 4, 8

3. Analysis of Feature Importance Using Random Forest

The main paper uses the Fisher-score to investigate the relative importance of individual features for predicting the status of individual traders (see Section 5.3 in the main paper). The Fisher-score ranking is univariate and cannot detect nonlinear patterns in the feature response relationship. To further elaborate on the relevance of individual features, we also examine feature importance using a random forest (RF) classifier. Feature importance scores extracted from tree-based ensemble classifiers are a popular way to quantify the relative impact of features on the response variable [22]. Figure 1 depicts the distribution of RF-based normalized importance scores for the first fifty features (ordered in terms of importance); the remainder being omitted to ensure readability. We highlight those features that have previously been identified as important by the Fisher-score.

Comparing Figure 1 and Table 2 in the main paper reveals differences between the variance adjusted comparison of group means, which the Fisher-score embodies, and the RF-based ranking. For example, the strongest feature according to that table, *ProfitxDur20*, does not appear in Figure 1 and the highest rank that a feature of the Fisher-score ranking achieves in Figure 1 is ten, as observed for the feature capturing a trader's average over the last twenty trades prior to the decision point. Interestingly, this feature, *PassAvgReturn20*, is the one that STX use in their hedging policy.

RF generates importance scores through comparing (out-of-bag) classification performance on the original data and that data after corrupting one feature through adding random noise. The magnitude of the performance decrease captures the importance of the corrupted feature [22]. This implies that RF assesses the importance of one feature vis-a-vis all other features, whereas the Fisher-score assesses one feature at a time. Given the different mechanism to measure importance, some differences between the RF and Fisher-score ranking are to be expected. It is still surprising that the most important features

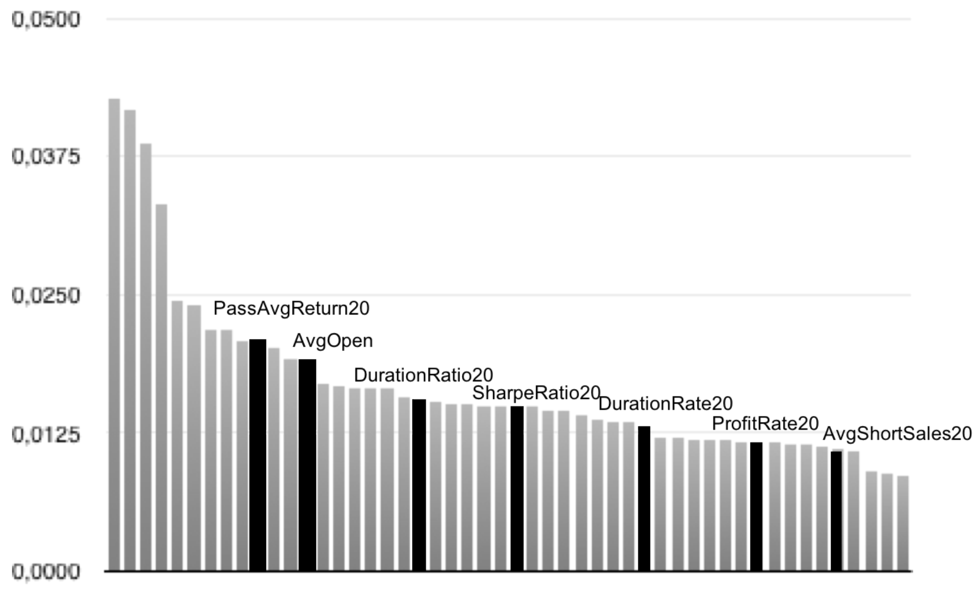

Figure 1: Normalized variable importance scores based on RF-classifier for the top 50 features. Dark color identifies features that also appear in the Fisher-score ranking in the main paper

of the latter receive relatively low ranks in Figure 1. An interpretation of this result is that it evidences intricate dependencies between the binary response and features, which the Fisher-score does not capture.

With respect to the complexity of the feature-response relationship, the distribution of importance scores in Figure 1 may be considered evidence of a set of three to four features being particularly strongly related to the response. We caution against this interpretation. The distributional shape is a consequence of the scaling of the y-axis, to ensure readability of the figure. The magnitude of importance scores is small, even for the left-most features. Therefore, importance differences between features (e.g., feature four and five) appear more substantial than they are. Recall that the scores capture the degree to which RF performance decreases if we corrupt one feature. Given the magnitude of importance scores, we interpret the results of Figure 1 as evidence of a low signal between the raw features and the future success of a trader. This emphasizes the trader classification task to be challenging. Even a powerful RF classifier, often observed to predict accurately [18, 20, 23], fails to identify strong dependencies among the raw features and the target. Low importance scores also question representativeness of the training data. This motivates our analysis whether a DNN, equipped with higher depth than RF, is able to learn more abstract, latent, features that enable predicting traders' future performance more accurately than conventional 'shallow' learners.

We complete the analysis of feature importance by aggregating importance scores across the main feature groups in Figure 2. The analysis offers insight as to the relative importance of different types of trader characteristics. The results displayed in Figure 2 agree with the views of STX dealing desk members. We find trader demographics and features in the markets & channels category to carry least weight, which reinforces the view that propensity for risk taking may be attributed to the competence and trading style rather than particular country of origin or gender. Past performance and trading discipline are most important for high risk trader identification, substantiating the claim that features capturing the professional behavior of traders are of primary value for the task at hand.

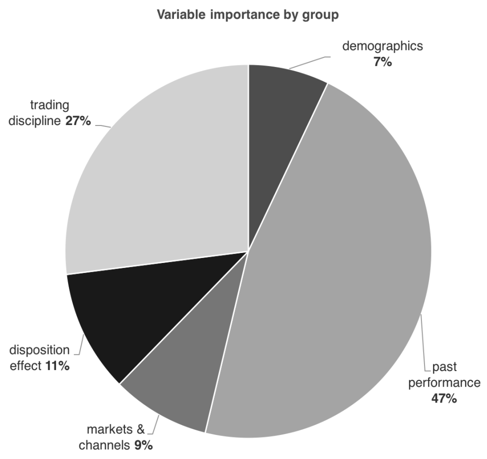

Figure 2: Analysis of group-level feature importance. The aggregation is performed by adding up the RF-based importance scores of all features belonging to the same group and normalizing group-level scores to sum to unity.

4. Robustness of the DNN With Respect to Random Weight Initialization

We train the DNN using stochastic gradient descent. Starting the minimization of the loss function from randomly initialized weights using Xavier initialization (see above), gradient descent will deliver different solutions depending on the random initial weights. Therefore, the performance of the DNN may vary with random initial weights. To examine the robustness of the DNN with respect to the initial weights, we repeat the initialization ten times, develop a DNN model on the training set, and assess its performance in terms of the AUC on the test set. More specifically, we first invoke the random number generator using a seed of 123. Using this seed, we draw ten pseudo-random integer numbers ranging from zero to 10,000. We store these numbers as seed values for the analysis of DNN robustness. That is, we loop over the ten random numbers and use the current number as seed value for the random number generator prior to training a DNN with randomly initialized weights. We then assess the performance of the resulting DNN in terms of the AUC. Table 3 reports corresponding results and suggests that the dependence of the DNN with respect to initial weights is not substantial.

5. Comparing the DNN to Other DL Models

The following subsections augment the empirical results presented in the main part of the paper. We explain in the main part of the paper that the ability to learn high level distributed representations from input data is not specific to the DNN we propose here. Prior literature credits the whole family of DL methods for this feature [17]. Therefore, we experiment with other popular DL methods and compare their performance in trader risk behavior forecasting to that of the proposed DNN. Corresponding results shed light on the effectiveness of the proposed DNN relative to other DL benchmarks and contribute

Table 3: Robustness of DNN Performance With Respect to Initial Weights

Seed	AUC
3582	0.821
1346	0.841
5218	0.833
7763	0.829
9785	0.815
7382	0.831
5857	0.846
96	0.828
6257	0.845
6782	0.830
Mean	**0.832**

additional insight to what extent unsupervised pre-training as well as other architectural choices we have made contribute to the performance of our DNN.

Considering encouraging results in the area of mortgage default prediction [24], the first DL benchmark we consider consists of a deep feed-forward network (DFNN) with more than one hidden layer. We also consider a convolutional neural network (CNN). While prior work has, to our best knowledge, not considered CNNs for risk analytics, CNNs have shown excellent results in other domains [25], which indicates that they represent a useful benchmark. Finally, using the sequence of trades per trader as a time-ordered input, we compare the proposed DNN to a recurrent neural network with long short-term memory cells (LSTM) [26]. Using the same performance indicators and statistical tests as in the comparison to ML benchmarks (Table 3 in the main paper), we report the results of the three DL methods together with those of our DNN in Table 4.

While results in the main paper are based on cross-validation to ensure robustness, comparisons to other DL approaches are undertaken using a simpler, computationally less demanding split-sampling approach. This approach involved partitioning the data sequentially into a training set (70%) for developing predictive models and a test set (30%) for assessing their accuracy. Trades from November 2003 to April 2013 entered the training set, whereas trades from May 2013 to July 2014 served as the hold-out test set. To address the class imbalance in the comparison, we bootstrapped the test set with different ratios of class '+1' trades ranging from 0.05, 0.1, ... 0.5, and drew 1000 bootstrap samples for each ratio. This approach enabled examining the performance of the DNN across different scenarios with varying degrees of class skew and increased robustness because it implied repeating the out-of-sample evaluation 11000 times on different (bootstrapped) test sets. At the same time, bootstrapping the test set is substantially less computationally demanding than using cross-validation and SMOTE; that is the approach we take in the main part of the paper.

The overall conclusion from Table 4 is that the other DL benchmarks do not perform as well as the proposed model. Our DNN consistently achieves the best performance across evaluation criteria and class ratios. Therefore, Table 4 supports the proposed DNN and its underlying topological choices (see Figure 3 in the main paper). In particular, none of the three DL benchmarks employs unsupervised pre-training. Therefore, the superior performance of the proposed DNN may be taken as evidence for the suitability of unsupervised pre-training.

However, we caution against over-emphasizing results of Table 4. DL methods are complex and require careful tuning to unfold their full potential. This paper focuses on one particular type of DNN and its potential to support decision-making in risk management. Performing a fully-comprehensive benchmark of several alternative complex DL models is beyond the scope of the paper. Accordingly, we do not claim superiority of the proposed DNN to deep, feed-forward networks, CNNs and other DL methods in general, and acknowledge that more elaborate tuning of corresponding approaches may give performance comparable to the DNN we employ.

Table 4: Comparison of Proposed DNN Against Other Deep Learning Benchmarks

Metrics		Bootstrap with different percentage of high risk clients										
	Classifiers	0.05	default	0.10	0.15	0.20	0.25	0.30	0.35	0.40	0.45	0.50
P&L (million)	Proposed DNN	22.50	18.30	14.92	7.43	-0.25	-7.84	-16.07	-23.28	-30.85	-38.70	-45.83
	DFNN	19.54	16.43	12.53	5.42	-2.34	-8.90	-18.43	-25.65	-32.54	-40.55	-52.21
	CNN	20.53	14.43	12.43	5.01	-4.53	-9.42	-19.43	-30.43	-35.62	-47.33	-54.45
	LSTM	21.09	15.01	14.2	3.54	-5.42	-9.21	-17.54	-26.75	-34.53	-45.76	-50.54
AUC	Proposed DNN	0.813	0.812	0.812	0.812	0.812	0.812	0.812	0.812	0.812	0.812	0.812
	FDNN	0.704	0.704	0.704	0.604	0.703	0.704	0.704	0.704	0.705	0.704	0.704
	CNN	0.793	0.793	0.793	0.793	0.791	0.793	0.793	0.793	0.793	0.793	0.793
	LSTM	0.745	0.745	0.745	0.745	0.746	0.745	0.743	0.744	0.745	0.745	0.745
F-Score	Proposed DNN	0.248	0.282	0.298	0.318	0.331	0.338	0.343	0.347	0.35	0.352	0.354
	DFNN	0.11	0.204	0.242	0.284	0.302	0.303	0.305	0.340	0.335	0.301	0.330
	CNN	0.094	0.032	0.199	0.242	0.303	0.306	0.312	0.302	0.329	0.339	0.302
	LSTM	0.081	0.225	0.205	0.209	0.321	0.312	0.309	0.301	0.321	0.305	0.329

Notes: We tune the hyper-parameters of the three DL benchmarks using grid-search in the same manner as the ML benchmarks. The hyper-parameters and search spaces we consider are as follows. *DFNN*: no. of hidden layers [2, 3, 4], no. of neurons per hidden layer [50; 200]. *CNN* no. of convolutional layers [2, 3, 4], filter size 3, max. pooling size = 2. *LSTM* no. of hidden layer [2, 3]. FDNN and CNNs use activation functions of type ReLu in the hidden layers.

References

[1] X. Glorot, Y. Bengio, Understanding the difficulty of training deep feedforward neural networks, in: International Conference on Artificial Intelligence and Statistics, 2010, pp. 249–256.

[2] V. Nair, G. E. Hinton, Rectified linear units improve restricted boltzmann machines, in: Proceedings of the 27th International Conference on Machine Learning (ICML-10), 2010, pp. 807–814.

[3] X. Glorot, A. Bordes, Y. Bengio, Deep sparse rectifier networks, in: Proceedings of the 14th International Conference on Artificial Intelligence and Statistics. JMLR W&CP Volume, Vol. 15, 2011, pp. 315–323.

[4] G. E. Hinton, R. R. Salakhutdinov, Reducing the dimensionality of data with neural networks, Science 313 (5786) (2006) 504–507.

[5] Y. Bengio, I. Goodfellow, A. Courville, Deep learning, MIT Press, 2016.

[6] S. Ioffe, C. Szegedy, Batch normalization: Accelerating deep network training by reducing internal covariate shift, arXiv preprint arXiv:1502.03167.

[7] W. A. Gardner, Learning characteristics of stochastic-gradient-descent algorithms: A general study, analysis, and critique, Signal Processing 6 (2) (1984) 113–133.

[8] O. Bousquet, L. Bottou, The tradeoffs of large scale learning, in: Advances in neural information processing systems, 2008, pp. 161–168.

[9] I. Sutskever, J. Martens, G. Dahl, G. Hinton, On the importance of initialization and momentum in deep learning, in: Proceedings of the 30th International Conference on Machine Learning (ICML-13), 2013, pp. 1139–1147.

[10] Y. Bengio, P. Simard, P. Frasconi, Learning long-term dependencies with gradient descent is difficult, Neural Networks, IEEE Transactions on 5 (2) (1994) 157–166.

[11] J. Duchi, E. Hazan, Y. Singer, Adaptive subgradient methods for online learning and stochastic optimization, The Journal of Machine Learning Research 12 (2011) 2121–2159.

[12] M. D. Zeiler, Adadelta: An adaptive learning rate method, arXiv preprint arXiv:1212.5701.

[13] L. Prechelt, Automatic early stopping using cross validation: quantifying the criteria, Neural Networks 11 (4) (1998) 761–767.

[14] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in Python, Journal of Machine Learning Research 12 (2011) 2825–2830.

[15] T. Fitzpatrick, C. Mues, An empirical comparison of classification algorithms for mortgage default prediction: evidence from a distressed mortgage market, European Journal of Operational Research 249 (2) (2016) 427–439.

[16] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, in: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, USENIX Association, 2012, pp. 2–2.

[17] Y. LeCun, Y. Bengio, G. Hinton, Deep learning, Nature 521 (7553) (2015) 436–444.

[18] S. Lessmann, B. Baesens, H.-V. Seow, L. C. Thomas, Benchmarking state-of-the-art classification algorithms for credit scoring: An update of research, European Journal of Operational Research 247 (1) (2015) 124–136.

[19] S. Finlay, Multiple classifier architectures and their application to credit risk assessment, European Journal of Operational Research 210 (2) (2011) 368–378.

[20] W. Verbeke, K. Dejaeger, D. Martens, J. Hur, B. Baesens, New insights into churn prediction in the telecommunication sector: A profit driven data mining approach, European Journal of Operational Research 218 (1) (2012) 211–229.

[21] Y. Bengio, Practical recommendations for gradient-based training of deep architectures, in: Neural Networks: Tricks of the Trade, Springer, 2012, pp. 437–478.

[22] T. Hastie, R. Tibshirani, J. H. Friedman, The Elements of Statistical Learning, 2nd Edition, Springer, New York, 2009.

[23] C. Krauss, X. A. Do, N. Huck, Deep neural networks, gradient-boosted trees, random forests: Statistical arbitrage on the S&P 500, European Journal of Operational Research 259 (2) (2017) 689–702.

[24] J. A. Sirignano, A. Sadhwani, K. Giesecke, Deep learning for mortgage risk (2016).
URL `https://people.stanford.edu/giesecke/`

[25] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, F. E. Alsaadi, A survey of deep neural network architectures and their applications, Neurocomputing 234 (2017) 11–26.

[26] T. Fischer, C. Krauss, Deep learning with long short-term memory networks for financial market predictions, European Journal of Operational Research 270 (2) (2018) 654–669.