Study and analysis of deep learning techniques for solving financial machine learning problems

Wendell João Castro de Ávila
Dep. de Ciência da Computação
Universidade Federal de Alfenas
Alfenas, Brazil
wendell.avila@sou.unifal-mg.edu.br

Ricardo Menezes Salgado
Dep. de Ciência da Computação
Universidade Federal de Alfenas
Alfenas, Brazil
ricardo.salgado@unifal-mg.edu.br

***Abstract*—This work documents the process and presents the techniques employed to develop deep learning models using a dataset containing over 2 million financial data observations. These models were developed using an incremental approach consisting of starting with simple models with default parameters and later isolating and improving their parts one step at a time. In each step of this process, various techniques with similar goals were compared, with the best performing technique being integrated into an existing model. With this approach, we aim to improve the performance of our models gradually at each step, with a constant focus on reducing the effects of overfitting. Additionally, this work can serve as a guide to help researchers working on similar problems. By suggesting resources that can be used and steps that can be followed in similar scenarios, this work can help narrow down the search for efficient financial machine learning models.**

***Index Terms*—Deep learning, finance, financial markets, machine learning.**

I. Introduction

This work aims to introduce and evaluate deep learning techniques that can be used to solve financial machine learning problems. For this purpose, a dataset provided by the financial machine learning competition Jane Street Market Prediction (JSMP) [1] was used.

This competition proposes the creation of machine learning models that can decide whether or not financial transactions should be performed, aiming for profit.

To do so, JSMP provided financial data consisting of a tabular dataset containing over 2 million data observations collected over the course of 500 days. In it, each observation represents a buying or selling opportunity for which a machine learning model must decide if it is worthwhile to perform it, considering the monetary return it can achieve.

This dataset was chosen for working with deep learning for two reasons:

1) We believe it is compatible with deep learning, given its large number of observations.
2) This dataset contains a set of 130 anonymized features: columns with generic names, with no information explaining what each feature and its values represent.

Anonymized features can make it difficult to use feature engineering and feature selection, techniques widely used in machine learning. Deep learning is capable of automatically identifying the best features that help predict desired results [2], not relying on feature selection and having an advantage over other machine learning algorithms.

To develop these models, we used an incremental approach consisting of starting with a simple model and isolating and improving its parts, allowing us to compare the performance of different approaches with similar goals and select the best to integrate an existing model.

This work also has focus on diminishing the effects of overfitting. When not properly accounted for, it often leads to failure in finance, producing false discoveries and promised outcomes that cannot be delivered [3, p. 11–12].

Following this approach, we aim to gradually improve the performance of our models, achieving better results after each step of including techniques in the models.

Other than documenting the findings in this specific application, this work can also help researchers working on similar financial machine learning problems by suggesting resources that can be used and steps that can be followed in similar scenarios, narrowing down the search for efficient models.

The content of this work is structured as follows: in section II, the theoretical framework regarding the techniques used is presented. Section III explores the dataset used in more detail. Section IV details the methodology used to develop and evaluate deep learning models. In section V, the results achieved using this methodology are presented and compared. Finally, section VI concludes this work with the final considerations.

II. Theoretical Framework

A. Machine Learning

A machine learning system is trained rather than explicitly programmed. It consists of using algorithmic solutions to find statistical relationships between a set of input data and its respective output data, approximating a function that can determine an output from a given input data.

A machine learning model can transform its input data into meaningful outputs, a process learned from exposure to known examples of inputs and outputs [4, p. 33], finding statistical structures in these examples that allow the system to come up with rules for automating the task [4, p. 32].

In addition to offering an automated solution, machine learning excels at being able to handle large and complex

multidimensional datasets to which the classical statistical approach would be impractical.

B. Artificial Neural Networks

Artificial neural networks, which serve as the basis for deep learning, are loosely inspired by the way neurons, our brain cells, process information in the brain [5, p. 165].

An artificial neural network is composed of units called perceptrons. These units are linked together by connections through which information flows from one unit to another. Each connection has an associated numerical weight, which determines the strength of the connection. Each unit then calculates a weighted sum of its inputs that is directed to an activation function—a function that applies a non-linear transformation to the sum. This nonlinear activation function gives the neural network the ability to represent most functions [6, p. 729].

These units are then grouped into layers where, in the more traditional feedforward network, each unit only receives inputs from units in the first preceding layer. After all layers executed this process, the output layer returns the predictions made and a cost function is used to evaluate them. The cost function evaluates how far off the predictions are compared to the true output data, allowing the weights of all units to be updated, decreasing the total error and bringing the predicted values closer to the true values.

Each unit, having different weights and inputs, analyzes different sections of the information with different priorities in order to approximate a function that maps input data to output data [5, p. 164]. This approximated function can used to make predictions for new input data for which the respective outputs are unknown.

C. Deep Learning

Deep learning is a subfield of machine learning that takes neural networks further by introducing multiple successive layers of units, having an emphasis on learning successive layers of meaningful representations in the data. Modern deep learning commonly involves dozens or even hundreds of successive layers of representation, all learned automatically by exposure to the training data [4, p. 35].

Deep learning excels at dealing with large amounts of data unlike any other machine learning algorithm. It also removes the need for feature engineering, as the model can learn and adjust its internal features jointly, greatly simplifying machine learning workflows [4, p. 35].

While neural networks have been researched for decades, deep learning only became a trend in machine learning in recent years, as it became more useful and powerful due to the availability of large amounts of data thanks to the internet and the popularization of graphics processing units (GPUs) for parallelizing computational tasks [4, p. 51] [5, p. 12].

D. GPU Acceleration

The training phase of deep learning is a slow and computationally intensive process, where each iteration requires the computation of many matrix multiplications. To accelerate this process, these operations can be executed on multiple computational units using parallelism.

GPUs can be used for this purpose, as they are not only a powerful graphics engine but also a highly parallel programmable processor featuring peak arithmetic and memory bandwidth that substantially outpaces its CPU counterpart [7, p. 879].

GPUs are built for different demands than CPUs, having to handle large amounts of data and hundreds of operations per input to be able to satisfy the demands of complex real-time applications. They are also perfectly suited for parallelism, relying on a large number of programmable processing cores and having a focus on throughput rather than latency [7, p. 879].

GPU's increase in programmability and capability allowed mapping a broad range of computationally demanding, complex problems to the GPU [7, p. 879]. One of these applications is deep learning, where its use in a machine learning competition in 2012 sparked the current interest in it [8].

E. Feature Engineering

Feature engineering is the process of using one's own expert knowledge about a specific domain to make the algorithm work better by applying nonlearned transformations to the data before training a machine learning model [4, p. 151].

With anonymized features, it is difficult to apply expert knowledge for feature engineering in a reliable way, as the true meaning of each feature can only be guessed.

F. Feature Selection

Feature selection consists of selecting an appropriate set of features that have desired properties for solving a particular problem. Designing the ideal feature space can incur a huge cost in terms of computational time or expert knowledge [2].

G. Underfitting, Overfitting, and Regularization

A good machine learning model should be able to generalize a problem well, performing well not only on the data it was trained but also on new inputs [5, p. 224].

To achieve that, machine learning models are evaluated not only for their ability to approximate the true outputs during training but also for their capacity to correctly predict outputs over inputs the training algorithm did not have access to the respective true outputs.

Two problems that could prevent a model from achieving good generalization are underfitting and overfitting.

Underfitting occurs when, during training, a model fails to achieve a low training error, being unable to identify structures in the data that connect input data to outputs and causing the model to perform below what would be optimal for solving the problem. This problem usually happens with models that are too simple for the complexity of the data.

Overfitting can happen when a model is overly complex for a given problem, being able to achieve low training error but having a high test error. What causes this discrepancy between

training error and test error is that the model tries to adapt to the training data in an increasingly precise way, specializing only in that data. This causes the model to lose generalization power and to have poor performance when presented with new data, different from those in which the algorithm has over-specialized.

While preventing underfitting is just a matter of developing better models, appropriate measures need to be taken to minimize overfitting while improving models. There are strategies explicitly designed to reduce the testing error, possibly making the training error worse, that can help a model achieve better generalization. These strategies are called regularization [5, p. 224].

H. Validation with Train and Test Split

To train a neural network with a good generalization power, we need to evaluate the model's performance not only on training data but also on new inputs.

One way of doing this is providing the model with two sets of data during training: a training set and a test set. The learning process is performed on the training set. The test set, however, is used for partial evaluation of the model on new inputs during training, evaluating the generalization performance of the model at the same time it is trained. These two sets must be defined so that each observation of the total data belongs to only one set, preventing leakage of information from one set to another.

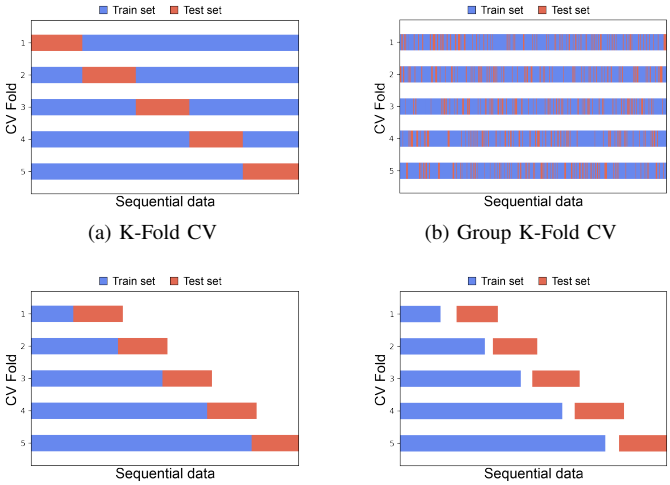
The purpose of the training procedure in a neural network is to make the training error decrease indefinitely. However, if this error decreases too much, the model could reach a state of overfitting, over-specializing in the training data, losing generalizing power, and causing performance on new inputs to be worse. A visual representation of this effect can be seen in [5, Fig. 5.3].

By monitoring the test error, or generalization error, calculated from the test set, we can identify the occurrence of overfitting and define the ideal moment to stop the training process before it overfits. Knowing the iteration of the training process where the generalization error tends to rise, we can train the model again and stop it before it reaches that iteration.

I. Early Stopping

A more efficient way of stopping the training procedure before the model overfits is an automated method called early stopping. This method allows training to be performed for a large and arbitrary number of iterations, with the end of the training process being defined by a stopping criterion rather than a specific number of iterations. This criterion is defined by an evaluation metric, calculated on the test set, that needs to be constantly improving to keep the training process going. When this metric fails to improve for a prolonged number of iterations, training is stopped.

It is also possible to define the patience for stopping: a number of iterations in which the training stoppage is delayed, allowing a possible rise in evaluation metrics following a drop.



(a) K-Fold CV (b) Group K-Fold CV

(c) Time Series Split (d) Purged K-Fold

Fig. 1. Visualization of how data is split using different CV strategies.

After the training procedure stops, the state of the model at the time it reached its peak is restored.

Due to its simplicity and unobtrusive nature, early stopping is the most commonly used regularization technique for deep learning, requiring no changes to the underlying training process or to the network structure [5, p. 245].

J. Cross-validation

Defining a single training set and a single test set is an approach that still leaves room for overfitting to occur. As we seek models that have a low test error, we may be selecting models that are only good at predicting for that particular test set, and not necessarily from new inputs in general. To have a more reliable test error estimate, cross-validation (CV) can be used.

The simplest variant of CV, called K-fold Cross-validation and illustrated in Fig. 1.a, consists of separating the data into k sets of similar size, called folds, where each observation belongs to only one of the sets. The model is then trained using the data from $k-1$ sets, while the remaining set is used for testing. This process is repeated until all k sets serve as the test set exactly once. At the end of the process, performance is evaluated by calculating the average of the errors found in each run [9].

Another variant, called Group K-fold Cross-validation, is used when there is a need to keep together observations pertaining to a group or category. This variant ensures that by defining a feature to be evaluated, observations that contain the same value for that feature will not be split between train and test sets. [9].

To handle sequential data, there is also the variant Time Series Split, illustrated in Fig. 1.c. This variant separates the data while maintaining the temporal order between individual observations, as well as ensuring that the test data will always be after the training data in the time sequence [9].

Prado [3, p. 104–105] warns of a deficiency in CV strategies that leads to failure when working with financial data. This issue is caused by the sequential correlation between rows in the dataset, since consecutive data points can be connected through information, causing information to be leaked from the training set to the test set leading to overfitting. To fix this, a cross-validation strategy called 'Purged K-fold' is proposed. This strategy, illustrated in Fig. 1.d., is an enhancement of Time Series Split that creates a 'gap' between training and test sets, defining a fixed number of observations that will not belong to either set, distancing the sets in the temporal order to minimize information leakage by sequential correlation [3, p. 105–110].

When using CV, we obtain as a result a number of trained models equal to the number of folds used to divide the data. While these models have the same structure, each one learned different information from different sections of the data.

K. Influence of Random Seeds on Results

Deep learning training algorithms rely on nondeterminism to improve model accuracy and training efficiency. This nondeterminism introduces variance in deep learning approaches, causing training runs with the same settings to produce different deep learning models with significantly different accuracies [10].

One way of reducing variance is setting fixed seeds: providing a number that controls how the pseudorandom algorithm used to generate random values will operate. This can make certain parts of the training process to be performed with the same settings, having a better comparison between multiple runs [10].

It is also important to note that selecting a seed that performed well in a specific validation set do not correspond to a good performance on new, unseen data in general. Random seeds are not a hyper-parameter to be optimized; selecting seeds that appear to produce better results will lead to optimistically biased performance measures. Instead, they can be used to reduce variance by averaging the results obtained in multiple runs with different seeds [11].

L. Imputation of Missing Values

Many datasets contain observations with unavailable information in some of the features.

Neural networks work exclusively with numerical values, so these missing spots in the data need to either be removed or imputed with some value.

A simple strategy that can be used to impute values in these missing spots consists of assigning the mean of each numerical feature to every missing value in that feature, or the most frequent value for categorical features. Another simple strategy is forward fill, where each row copies values from the first preceding row with a valid value.

More advanced strategies based on multivariate imputation can also be used, such as MICE [12] or scikit-learn's IterativeImputer [13].

M. Batch Normalization

During training, the distribution of input values in layers is always changing as the parameters and weights of previous layers change. This makes training slower by requiring lower learning rates and careful parameter initialization [14].

By normalizing inputs as they enter each layer, adjusting values measured on different scales to a common scale, 'batch normalization' can help speed up and stabilize training in deep neural networks, also making them less prone to the effects of parameter initialization and providing some regularization [14].

N. Dropout

Overfitting can be reduced with the use of 'dropout', a technique that works by randomly deactivating some hidden units of each layer during training. This essentially simulates different models within the same model, as the model will be forced to learn different structures in the data at each iteration of training due to parts of the information being missing [15].

By learning different structures at each iteration, the model can be less reliant on specific structures found in the training data and instead use more generalist structures that would work in multiple scenarios, increasing the model's generalization power.

O. Label Smoothing

Label smoothing can prevent a model from becoming overconfident in its predictions. It does so by turning hard labels, like zeros and ones in a classification problem, into soft labels: percentages that are not 100% for either class. For example, each observation with a target value of one could be considered 95% one and 5% zero. It is still not clear why label smoothing works, but its use has shown remarkable results in many applications [16].

P. Hyper-parameter Tuning and Hyperopt

There are many parameters in a machine learning model whose values are not learned through training. These parameters, called hyper-parameters, have a great impact on the learning process and need to be fine-tuned when building models.

Manually defining values for things such as the number of layers and units in each layer, regularization parameters, etc., is not a viable option in deep learning, as models often have dozens and even hundreds of hyper-parameters. Instead, we can use an optimizer to do this tiresome work.

An optimizer such as Hyperopt [17] can test thousands of value combinations among predefined ranges for multiple hyper-parameters at once, training models with these values and ranking them by how they perform.

Q. Dimensionality Reduction

Data with an excessive number of dimensions can be harder to work with in machine learning, as using more features can cause an increase in data sparsity and require a bigger number of computational resources. This phenomenon is called the

'curse of dimensionality', coined by [18]. Ideally, data should only have a dimensionality corresponding to the intrinsic dimensionality of the data, containing only the minimum number of dimensions needed to observe desired properties within the data [19].

Dimensionality reduction can be used to transform high-dimensional data into a meaningful representation of reduced dimensionality, diminishing the effects of the curse of dimensionality.

R. Principal Component Analysis

Principal Component Analysis (PCA) is an unsupervised dimensionality reduction technique that can simplify the complexity of high-dimensional data while retaining trends and patterns. It does so by geometrically projecting them onto lower dimensions called principal components, aiming to find the best summary of the data using a limited number of principal components [20].

S. Autoencoders

The autoencoder algorithm is another unsupervised dimensionality reduction method implemented using artificial neural networks. It aims to learn a compressed representation of its inputs [21].

Autoencoders are typically composed of three layers: the first, called encoder, is trained to compress data into a central layer that is smaller than the encoder, while the last layer, called decoder, is trained to decompress the data from the central layer into a state that is as close as possible to the input data [22].

With the central layer being restricted to a number of units smaller than the number of original input nodes, autoencoders produce a compressed representation of the input data that can convey the same information present in the original data, achieving the desired dimensionality reduction effect [23].

T. Ensemble and Bagging

Ensemble consists of combining predictions of different individually trained models, producing a single predictor that takes advantage of the structures learned by each model in different parts of the input space [24].

One of the most used methods for ensembling models is called bootstrap aggregation, also known as bagging. It works by calculating the simple average of the predictions found by each individual model, being able to reduce variance and bias by using a majority voting approach rather than relying on a single model that could be underperforming in certain parts of the input space or experiencing overfitting [3, p. 94–98].

III. Dataset

The dataset provided by the competition, according to [25], contains 2 390 491 rows of financial data collected over the course of 500 days. Each row representing a business opportunity for which an *action* value has to be predicted: one to perform the trade and zero to refuse it.

This dataset is sequential, with data observations arranged in a temporal order. It contains a set of 130 anonymized features, meaning that nothing is known about what each feature and its values represent, as they are not named with descriptive labels. Other columns provided and properly identified are detailed in Table I.

TABLE I
Training Dataset

Feature	Description
date	Indicates the day on which the trade occurred. Assumes integer values between 0 and 499.
weight	Numerical weight of each trade. Assumes positive real values.
resp	Return of each trade. Assumes real values.
resp_{1,2,3,4}	Same as *resp*, but at different time horizons.
feature_ {0,1,...,129}	Anonymized features. Each feature assumes different numerical ranges.
ts_id	A sequential unique identifier. Assumes positive integer values.
action	Field that needs to be added to the dataset, associating each trade to an action. Must assume the values 0 or 1.

From these 130 features, 88 of them contain missing values that need to be dealt with.

One particular property of this dataset confirmed by the organization of the competition is that, while it contains chronologically ordered data, its rows come from multiple different assets [26]. This means that each row might not have a relationship with other adjacent rows, and due to the anonymized features it is not possible to separate these rows by asset either, essentially making each trading opportunity an independent event.

In addition to this training dataset, the competition also detailed how its evaluation dataset, used by the platform to calculate scores and rank models, is structured. This dataset is served by the platform row-by-row on server-side when submitting models. Access to this dataset was not made available to participants.

This evaluation dataset differs from the training dataset detailed in Table I only by the absence of the fields *resp* and *resp_{1,2,3,4}* [25].

IV. Methodology

This work adopts a quantitative and experimental approach consisting of testing and evaluating different deep learning and machine learning techniques in an incremental way. By comparing the performance of various techniques with similar goals, we select those best fitted for the problem at hand. At the end of each step, the chosen technique is integrated into an existing model formed by the techniques selected in the previous steps.

By starting with a simple model, we can isolate and improve its parts, aiming to gradually improve its performance as we progress. This approach is also needed due to the fact

that testing every combination of every technique with every possible hyper-parameter value in an extensive combinatorial search is unfeasible when working with deep learning.

The metric used to evaluate the results is the 'utility score', proposed by the competition [1] and defined as follows:

Each trade j has an associated *weight* and *resp*, which represents a return. For each date i:

$$p_i = \sum_j (weight_{ij} * resp_{ij} * action_{ij}) \tag{1}$$

$$t = \frac{\sum p_i}{\sqrt{\sum p_i^2}} * \sqrt{\frac{250}{|i|}} \tag{2}$$

where $|i|$ is the number of unique dates in the test set. The utility is then defined as:

$$\text{utility score} = \min(\max(t, 0), 6) \sum p_i \tag{3}$$

This work was concluded many months after the end of the submission phase of the JSMP competition. This rendered us unable to use the official evaluation from the competition, as the evaluation dataset was not made available. Instead, we defined a validation set to simulate the official evaluation dataset. This validation set is composed of the last 50 days of data in the training dataset, while the remaining 450 days were used for training. Like in the evaluation set, the fields *resp_{1,2,3,4}* and *resp* were removed, with the latter being used to calculate the utility score after inference.

The implementation of the code used in this work was made using Python and its libraries, with the execution being performed on Kaggle's cloud computing environment [27] using GPU acceleration.

For the implementation and execution of neural networks, the Keras library (version 2.6.0) was used. Among other neural network libraries available, Keras was chosen for its ease of use and ease of learning. Other libraries used include numpy (version 1.21.6) and pandas (version 1.3.5), used for data processing, sci-kit learn (version 0.23.2), which provides various machine learning tools, matplotlib (version 3.5.2), for automated plotting of graphs and images, and Hyperopt (version 0.2.7), for hyper-parameter tuning. All code used in this work is available for public access on Kaggle [28] and GitHub [29].

Training was performed using a 5-fold CV strategy called Purged Group Time Series Split, a simple modification of Purged K-fold that includes the group separation from Group K-fold. This was used to keep data from the same day together, as having intra-day information split between train and test sets could cause information leakage. A gap of 20 days between train and test sets was used. To combine the predictions of the five models trained in the 5-fold CV, the weighted average proposed by [30, eq. (3)] was used, giving higher importance to models trained in more recent data as well as accounting for the uneven sizes of training sets seen in Fig. 1.d. To stop each training procedure, early stopping was used with patience set to 12 iterations.

We started by evaluating different types of supervised deep learning models. With the training dataset containing observations from multiple assets, we opted not to use Long Short-term Memory (LSTM) networks—widely used for sequential data [31]—as this property of the data would defeat the purpose of recurrent networks. Instead, multilayer perceptron networks were used.

As seen in section III, the column responsible for determining which trades are performed is not included in the training dataset. It can easily be set by doing a simple transformation over the *resp* column, with negative values of *resp* translating to zero and positive values translating to one. With this newly set binary column, it is possible to work in a classification model that can assign trades the values zero and one. Alternatively, we could instead build a model to predict the *resp* field itself, later converting the predictions into zeros and ones to get the *action* column.

Other possibility that this dataset gives us is to use the alternative *resp* fields: *resp_1*, *resp_2*, *resp_3*, and *resp_4*. Instead of predicting a single *action* value based on the *resp* field, we could instead predict 5 *action* values based in the 5 *resp* fields available, taking their average to produce a final answer.

These starter models were made using the hyper-parameters listed in Table II. The majority of them are the most commonly used for the respective type of model, such as the Adam optimizer [32] and the Rectified Linear Unit (ReLU) activation function [33], with exception being the large batch size used to speed up training due to the large size of the dataset.

As stated by [5, p. 427], manually selecting hyper-parameters can work well when the user has a good starting point, such as information given by others who worked on the same type of application, or when the user has months or years of experience in exploring hyper-parameters for similar tasks. As we do not possess such starting points, we took the liberty of selecting an arbitrary network topology with three hidden layers, each with a number of units equal to the number of units in the input layer. This topology, as well as other hyper-parameters, will later be improved using automated methods for finding better hyper-parameters.

We opted not to remove any observation or feature due to missing data, as deep neural networks are favored by large amounts of data. Instead, imputation of missing values was performed using two simple strategies: mean and forward fill. More sophisticated strategies could not be used, as these would require a huge amount of computational resources to operate in such a large dataset.

Additionally, a missing indicator variant was tested for both strategies. This variant consists of adding a new binary feature to the dataset for each original feature with missing values. This new feature assumes the value one if the feature it was based on had a missing value in that row, and zero otherwise. This variant is used to keep a record of missing values after imputation, allowing the model to use the *missingness* of data as information.

TABLE II
MODEL SETTINGS

	Model type	
	Classification	Regression
Number of layers	3	3
Units in each layer	130;130;130	130;130;130
Learning rate	0.001	0.001
Activation function	ReLU	ReLU
Batch size	4096	4096
Optimizer	Adam	Adam
Loss function	Binary Cross-entropy	Mean Squared Error
Early stopping metric	Maximize AUC	Minimize Mean Squared Error
Missing value imputation strategy	Mean	Mean

For regularization, three techniques were tested both individually and in conjunction: batch normalization, dropout, and label smoothing.

To tune hyper-parameters, the bayesian optimizer Hyperopt was used. It allows us to set a custom objective function to be minimized, making it possible to use any kind of evaluation metric. Two different objective functions were used for testing hyper-parameters:

1) The negative weighted average of the five early stopping metrics achieved in a 5-fold CV.
2) Utility score obtained from predictions subtracted from the maximum utility score calculated using the true values. This objective function also uses a 5-fold CV.

The tuning process was split into three steps, with the values found in each step being fixed to perform the following step. This choice was made to make the search space smaller, allowing the search algorithm to test a bigger number of combinations in each step. These steps are:

1) Tune activation functions, testing the widely used ReLU activation and the Swish activation, an activation function proposed by [34] to replace ReLU that showed improvement on deep networks applied to a variety of domains.
2) Tune different values for number of layers and number of units in each layer.
3) Tune regularization hyper-parameters such as dropout rates and label smoothing factor.

The ranges and options tested for each hyper-parameter can be seen in Table III.

For dimensionality reduction, two techniques were evaluated: PCA and Autoencoders.

TABLE III
HYPER-PARAMETER TUNING - RANGES AND OPTIONS

Hyper-parameter	**Value ranges/options**
Activation function	[relu, swish]
Number of layers	$3 \leq x \leq 5, x \in \mathbb{N}$
Units in each layer	$32 \leq x \leq 1024, x \in \mathbb{N}$
Dropout rate - input layer	$0.0 \leq x \leq 0.2, x \in \mathbb{R}$
Dropout rate - hidden layers	$0.2 \leq x \leq 0.5, x \in \mathbb{R}$
Label smoothing factor	$0.0 \leq x \leq 0.5, x \in \mathbb{R}$

To decide the number of principal components to use as a limit to PCA, a 'scree plot' was made, suggested by [35]. This plot can help us visualize the contribution of each subsequent principal component to the summarization of the original data.

The autoencoder used in this work is not a separate model to preprocess the data. Instead, it can turn a classification model into a 'deep bottleneck classifier', with the encoder added to the existing structure of a classification model as a layer that goes after the input layer and before the first hidden layer. This was proposed by [36] as a way to introduce supervision in autoencoders by taking advantage of the backpropagation coming from the supervised classification model to finetune its reconstruction error.

Lastly, bagging was used for ensembling models. To select more models for the ensemble, hyper-parameter tuning was performed again to select two more models, totaling 3 models with similar performance.

V. RESULTS

This section presents and compares the performance of the techniques defined in section IV using the utility scores calculated over the validation set using the predictions of each model. The maximum utility score that can be achieved with this validation set, calculated using its true values, is 20 083.50.

To remedy the effects of random seeds on results and increase the statistical validity of this work, five executions were performed, each with a different fixed seed. These five different seeds are the same for every analysis—zero, one, two, three, and four—and were set for both training and validation.

Starting with model types, we can see in Table IV that regression showed considerably worse results than classification, with the multitarget approach achieving slightly better scores than the simple classification approach on average. With these results, the multitarget classification model was selected in this step.

Next, imputation methods were evaluated using the multitarget classification model selected in the previous step. Results in Table V show a slight increase in the average utility score by using missing indicator in conjunction with mean, while forward fill achieved worse results in both variants. Due to this

TABLE IV
RESULTS - MODEL TYPE

Model Type	Utility score					
	Seed 0	Seed 1	Seed 2	Seed 3	Seed 4	Average
Regression	144.36	141.85	261.66	96.92	198.78	168.71
Classification	717.30	477.30	662.00	477.67	573.62	581.58
Multitarget classification	581.33	664.92	378.72	778.63	597.43	600.21

increase in scores, mean with missing indicator was selected for missing value imputation.

TABLE V
RESULTS - IMPUTATION OF MISSING VALUES

Imputation Strategy	Utility score					
	Seed 0	Seed 1	Seed 2	Seed 3	Seed 4	Average
Mean	648.78	555.43	500.95	767.29	702.62	635.01
Forward fill	814.34	720.76	382.12	750.80	487.92	631.19
Mean with missing indicator	686.49	655.70	484.19	745.72	639.70	642.36
Forward fill with missing indicator	675.19	742.78	420.76	820.62	433.30	618.53

Regularization techniques were evaluated individually and in pairs. In [15], dropout was used by randomly deactivating 50% of units in hidden layers and 20% of units in the input layer. For this initial analysis, a lower rate of dropout was used, with 30% for hidden layers and 10% for input layers. For label smoothing, a small factor of 0.1 was used to create the soft labels.

Dropout stands out with the biggest contribution to an increase in the utility score, as can be seen in Table VI. Batch normalization and label smoothing, while showing little improvement as standalone regularization strategies, managed to increase dropout's scores further when coupled with it. For this reason, all three regularization techniques were kept in the model.

The hyper-parameters found after following the 3-step hyper-parameter tuning with both objective functions can be seen in Table VII, while its results are listed in Table VIII.

Unfortunately, due to the large size of the data and models used, a small limit of combinations tested in each execution had to be set to avoid out-of-memory errors. A limit of 30, 20 and 15 combinations were used for models with three, four, and five hidden layers, respectively, effectively limiting the algorithm's capability of finding better models.

TABLE VI
RESULTS - REGULARIZATION

Regularization Strategy	Utility score					
	Seed 0	Seed 1	Seed 2	Seed 3	Seed 4	Average
None	575.56	677.33	402.88	550.92	530.29	547.40
Batch normalization	478.22	739.49	548.07	543.913	514.91	564.92
Dropout	706.221	756.425	768.530	831.643	770.701	766.70
Label smoothing	665.77	538.62	514.37	575.48	497.55	558.35
Batch normalization + dropout	890.47	831.44	968.01	900.56	865.70	891.23
Batch normalization + label smoothing	444.16	681.13	545.59	600.77	453.57	545.04
Dropout + label smoothing	810.35	805.46	868.17	793.73	804.24	816.39
All 3 strategies	913.62	807.21	951.89	835.60	830.83	867.83

The two objective functions achieved similar results with three and four hidden layers. When compared with the previous model, both achieved slightly worse scores with three layers and fairly worse scores with four layers. The second objective function, however, managed to find a better model with five layers. This was the model used in the following step.

Starting with PCA for dimensionality reduction, the scree plot in Fig. 2 shows that less than 100 principal components are enough to achieve maximum explained variance, meaning that less than 100 principal components are enough to explain the original data in its entirety. With this analysis, 80 principal components were used.

With the results presented in Table IX, the autoencoder approach was chosen for dimensionality reduction, as it managed to improve scores compared to an execution without dimensionality reduction, while PCA achieved worse scores.

The hyper-parameters used for the three models used for bagging can be seen in Table X, while the results are listed in Table XI.

These results show that every ensemble of two models managed to outperform its individual models on average, with the 3-model ensemble also outperforming any 2-model ensemble.

Finally, all models trained were combined to form a single predictor, averaging the predictions of every model trained in each of the five folds of CV over five different seeds and three

TABLE VII
HYPER-PARAMETERS FOUND USING HYPER-PARAMETER TUNING

Hyper-parameter	**Objective function 1**			**Objective function 2**		
	Model 1	**Model 2**	**Model 3**	**Model 1**	**Model 2**	**Model 3**
Activation function	swish	swish	swish	swish	swish	swish
Number of hidden layers	3	4	5	3	4	5
Units in hidden layer 1	560	256	272	592	496	208
Units in hidden layer 2	592	288	1008	320	512	48
Units in hidden layer 3	1008	720	336	880	400	112
Units in hidden layer 4	-	240	560	-	112	848
Units in hidden layer 5	-	-	672	-	-	624
Dropout rate - input layer	0.0437	0.1190	0.1301	0.0562	0.0102	0.1554
Dropout rate - hidden layer 1	0.4622	0.2368	0.2007	0.4362	0.3626	0.4770
Dropout rate - hidden layer 2	0.3821	0.2059	0.3748	0.3752	0.2640	0.2004
Dropout rate - hidden layer 3	0.3044	0.3489	0.2981	0.3518	0.2080	0.4212
Dropout rate - hidden layer 4	-	0.3263	0.3524	-	0.2258	0.4607
Dropout rate - hidden layer 5	-	-	0.3143	-	-	0.4318
Label smoothing factor	0.1579	0.4483	0.2414	0.0704	0.1336	0.4721

TABLE VIII
RESULTS - HYPER-PARAMETER TUNING

Model	**Utility score**					
	Seed 0	**Seed 1**	**Seed 2**	**Seed 3**	**Seed 4**	**Average**
Previous model	896.15	786.61	819.53	844.59	805.06	830.39
Objective 1 Model 1	763.27	777.06	872.09	837.36	839.40	817.83
Objective 1 Model 2	729.94	766.64	822.54	573.39	718.72	722.25
Objective 1 Model 3	682.48	733.52	796.43	697.23	838.77	749.68
Objective 2 Model 1	775.05	808.22	819.60	841.29	841.87	817.20
Objective 2 Model 2	795.30	752.18	682.41	656.94	730.12	723.39
Objective 2 Model 3	901.96	871.85	922.16	804.22	846.92	869.42

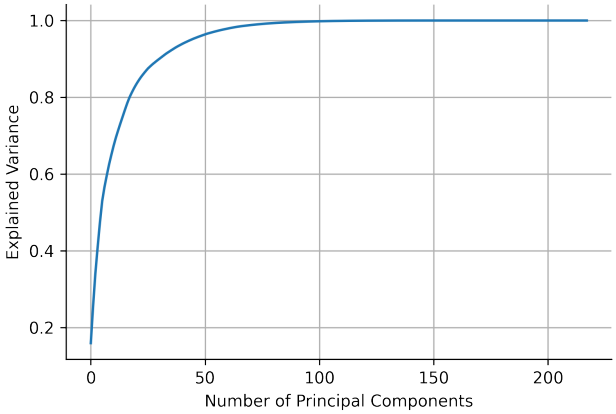


Fig. 2. PCA - Scree plot

TABLE IX
RESULTS - DIMENSIONALITY REDUCTION

Reduction Technique	**Utility score**					
	Seed 0	**Seed 1**	**Seed 2**	**Seed 3**	**Seed 4**	**Average**
No reduction	904.03	881.20	909.73	810.14	818.15	864.65
PCA	844.43	753.18	769.41	804.06	790.05	792.23
Autoencoder	801.69	938.18	922.18	958.51	761.89	876.49

different model structures. The utility scores achieved by the final bagging ensemble can be seen in Table XII.

VI. CONCLUSION

Comparing the utility score achieved by the first classification model evaluated in Table IV with the final 3-model 5-seed 5-fold classification ensemble, an increase of roughly 54% in the utility score can be seen. This increase is even more prominent when comparing the ensemble with the regression

TABLE X
ENSEMBLE - MODELS USED

Hyper-parameter	Model 1	Model 2	Model 3
Activation function	swish	swish	swish
Number of hidden layers	5	5	5
Units in hidden layer 1	208	240	416
Units in hidden layer 2	48	352	504
Units in hidden layer 3	112	796	256
Units in hidden layer 4	848	176	568
Units in hidden layer 5	624	448	368
Dropout rate - input layer	0.1554	0.1302	0.1747
Dropout rate - hidden layer 1	0.4470	0.2612	0.4661
Dropout rate - hidden layer 2	0.2004	0.4328	0.4207
Dropout rate - hidden layer 3	0.4212	0.4263	0.2335
Dropout rate - hidden layer 4	0.4607	0.4495	0.4132
Dropout rate - hidden layer 5	0.4318	0.3094	0.4563
Label smoothing factor	0.4721	0.4746	0.3964

TABLE XI
RESULTS - BAGGING

Model Ensemble	Utility score					
	Seed 0	Seed 1	Seed 2	Seed 3	Seed 4	Average
Model 1	829.96	889.57	749.19	774.30	815.01	811.61
Model 2	992.72	1062.25	682.31	636.83	750.68	824.96
Model 3	824.81	724.76	738.81	793.36	966.39	809.62
Model 1 + Model 2	902.39	980.16	776.86	672.77	814.04	829.24
Model 1 + Model 3	816.50	856.63	786.16	797.89	903.04	832.04
Model 2 + Model 3	884.08	844.88	814.29	796.01	880.61	843.97
All 3 Models	889.76	895.29	794.86	795.58	853.35	845.77

TABLE XII
RESULTS - BAGGING WITH ALL MODELS

Models x 5 seeds x 5 folds	Utility score
Model 1 (Total 25 models)	854.49
Model 2 (Total 25 models)	842.76
Model 3 (Total 25 models)	793.15
Models 1 + 2 (Total 50 models)	888.23
Models 1 + 3 (Total 50 models)	883.49
Models 2 + 3 (Total 50 models)	815.42
Models 1 + 2 + 3 (Total 75 models)	895.62

model of Table IV, the first to be discarded, with an increase of 430.87%.

By following the incremental approach with the evaluation method defined in this work and using the techniques presented here, we managed to gradually improve the model's predicting and generalization power, increasing the utility score at each step and achieving a final score that corresponds to 4.46% of the maximum utility score for the validation set used.

We believe that this approach can be used to solve similar financial machine learning problems with the same degree of success. By evaluating techniques incrementally, selecting those better fitted for each application, and having the reduction of overfitting as a key objective, reliable deep learning models with great predicting and generalization capabilities can be developed.

REFERENCES

[1] Jane Street Group, "*Jane Street Market Prediction*." kaggle.com. https://www.kaggle.com/c/jane-street-market-prediction/overview (accessed May 5, 2022)

[2] L. Arnold, S. Rebecchi, S. Chevallier, and H. Paugam-Moisy, "An introduction to deep learning," in *Proc. European Symp. on Artificial Neural Networks*, Brussels, 2011, pp. 477–488.

[3] M. L. de Prado, *Advances in Financial Machine Learning*, Hoboken, NJ: John Wiley & Sons, Inc., 2018.

[4] F. Chollet, *Deep learning with Python*, New York: Manning Publications, 2018.

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

[6] S. Russel and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2010.

[7] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[8] "From not working to neural networking," The Economist, June 25th 2016 ed., June 2016.

[9] scikit-learn, "*Cross-validation: evaluating estimator performance*." scikit-learn.org. https://scikit-learn.org/stable/modules/cross_validation.html (accessed May 20, 2022)

[10] H. V. Pham et al., "Problems and Opportunities in Training Deep Learning Software Systems: An Analysis of Variance," in *35th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*, 2020, pp. 771–783.

[11] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep Reinforcement Learning that Matters," in *Proc. of the 32nd AAAI Conf. on Artificial Intelligence*, New Orleans, 2018, pp. 3207–3214. doi:10.1609/aaai.v32i1.11694.

[12] S. van Buuren and K. Groothuis-Oudshoorn, "mice: Multivariate Imputation by Chained Equations in R," *Journal of Statistical Software*, vol. 45, no. 3, pp. 1-–67.

[13] scikit-learn, "*Imputation of missing values*." scikit-learn.org. https://scikit-learn.org/stable/modules/impute.html#iterative-imputer (accessed August 8, 2022)

[14] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proc. of the 32nd Int. Conf. on Machine Learning (ICML)*, Lille, France, Jul. 2015, pp. 448—456. doi:10.5555/3045118.3045167.

[15] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. "Improving neural networks by preventing co-adaptation of feature detectors," Jul. 2012.

[16] R. Müller, S. Kornblith, and G. Hinton, "When does label smoothing help?," in *33rd Conf. on Neural Information Processing Systems (NeurIPS)*, Vancouver, Canada, 2019.

[17] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Proc. of the 24th Int. Conf. on Neural Information Processing Systems*, Granada, Spain, Dec. 2011, pp. 2546–2554.

[18] R. Bellman, *Dynamic Programming*, New York: Princeton University Press, 1957.

[19] L. van der Maaten, E. Postma, and J. van den Herik "Dimensionality Reduction: A Comparative Review," *Journal of Machine Learning Research - JMLR*, vol. 10, pp. 66–71, Jan. 2007.

[20] J. Lever, M. Krzywinski, and N. Altman, "Principal component analysis," *Nature Methods*, vol. 14, pp. 641–642, 2017, doi:10.1038/nmeth.4346.

[21] W. Wang, Y. Huang, Y. Wang, and L. Wang, "Generalized autoencoder: A neural network framework for dimensionality reduction," in *Proc. of the 2014 IEEE Conf. Comput. Vis. Pattern Recog. Workshops*, 2014, pp. 496–503.

[22] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986, doi:10.1038/323533a0.

[23] Y. Wang, H. Yao, and S. Zhao, "Auto-encoder based dimensionality reduction," *Neurocomputing*, vol. 184, no. C, pp. 232–242, Apr. 2016, doi:10.1016/j.neucom.2015.08.104.

[24] D. Opitz and R. Maclin, "Popular ensemble methods: An empirical study," *Journal of Artificial Intelligence Research*, vol. 11, no. 1, pp. 169–198, July 1999.

[25] Jane Street Group, "*Jane Street Market Prediction - Data*." kaggle.com. https://www.kaggle.com/c/jane-street-market-prediction/data (accessed Mar 5, 2021)

[26] Jane Street Group, "*Discussion - Are the observations for a single instrument or different instruments?*." kaggle.com. https://www.kaggle.com/c/jane-street-market-prediction/discussion/199923#1094154 (accessed May 30, 2022)

[27] Kaggle, "*Notebooks Documentation*." kaggle.com. https://www.kaggle.com/docs/notebooks (accessed Mar 7, 2021)

[28] W. J. C. de Ávila, Aug. 2022, "*JaneStreet - Index*," Kaggle. [Online] Available: https://www.kaggle.com/wendellavila/janestreet-index/

[29] W. J. C. de Ávila, Aug. 2022, "*JSMP Notebooks*," GitHub. [Online] Available: https://github.com/wendellavila/JSMP-Notebooks

[30] J. P. Donate, P. Cortez, G. G. Sánchez, and A. S. de Miguel, "Time series forecasting using a weighted cross-validation evolutionary artificial neural network ensemble," *Neurocomputing*, vol. 109, pp 27–32, June 2013.

[31] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A search space odyssey," *IEEE transactions on neural networks and learning systems*, vol. 28, Mar. 2015. doi:10.1109/TNNLS.2016.2582924.

[32] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic optimization," in *Int. Conf. on Learning Representations (ICLR)*, 2015.

[33] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, Jan. 2015, doi:10.1016/j.neunet.2014.09.003.

[34] P. Ramachandran, B. Zoph, and Q. V. Le, "Swish: A self-gated activation function," Oct. 2017.

[35] L. H. Nguyen and S. Holmes, "Ten quick tips for effective dimensionality reduction," *PLOS Computational Biology*, vol. 15, June 2019, doi:10.1371/journal.pcbi.1006907.

[36] E. Parviainen, "Deep bottleneck classifiers in supervised dimension reduction," in *Proc. of the 20th Int. Conf. on Artificial Neural Networks: Part III*, pp.1–10, Sep. 2010.