Learning and Nonlinear Models - Journal of the Brazilian Society on Computational Intelligence (SBIC), Vol. 21, Iss. 2, pp. 43-54, 2023

STUDY AND ANALYSIS OF DEEP LEARNING TECHNIQUES FOR SOLVING FINANCIAL PROBLEMS

Wendell J. C. de Avila
Ricardo M. Salgado
Universidade Federal de Alfenas
wendell.avila@sou.unifal- mg.edu.br, ricardo.salgado@unifal- mg.edu.br.

Abstract – Financial markets are competitive environments influenced by several variables and sectors. Wrong decisions can compromise several areas and cause chain reactions that could disrupt various sectors of the economy. In recent years, intelligent models have been used as tools to aid decision-making in financial markets. Deep learning models stand out among them, as they can achieve good generalization with large datasets. The main goal of this paper is to introduce and evaluate deep learning for solving financial problems. We document the process and present the techniques employed to develop models using a dataset containing over 2 million financial data observations. We believe this paper could guide researchers working on similar problems by suggesting resources that can be used and steps that can be followed in similar scenarios, narrowing down the search for efficient financial machine learning models.
Keywords – Deep learning, machine learning, financial markets, finance.

1. INTRODUCTION

Financial markets are very complex and volatile adaptive systems that are constantly changing and being influenced by many external factors, such as news and political events. Making informed trading decisions requires taking a huge number of variables into account, inevitably making investors overwhelmed by an amount of information no human could process. This can lead to behavioral biases such as loss aversion, overconfidence, and overreaction, making investors rely on their personal experience and beliefs rather than on insights obtained from data. This behavior often leads to biased trading decisions that are indistinguishable from bets.

In recent years, research on the use of deep learning in finance has increased significantly, enabled by hardware advancements that made it viable for solving problems that rely on large amounts of data. Prediction tasks on financial data, however, are very difficult to accomplish. The non-stationary aspect of these markets makes solutions stop working when the environment changes, and market volatility makes it impossible to predict with certainty the profitability of any given trade.

The development and improvement of these algorithmic solutions could aid human investors in their decision-making process by providing insights from data that can be used to develop rational investment strategies. In the long run, large-scale use of these models could benefit markets by improving efficiency and productivity, as well as enhancing the quality of services offered to customers. This would also bring markets closer to a state of 'fairness', in which buyers and sellers would have all the agency and information needed to make rational trading decisions, positioning products in a 'fair' value range that would reduce volatility and speculation.

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

First, it contains more than 2 million observations. We believe it is compatible with deep learning, given its large size.

Second, this dataset contains a set of 130 anonymized features, making it difficult to use feature engineering and feature selection. Deep learning is capable of overcoming this limitation by automatically identifying the best features that help predict desired results [2].

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 of them to integrate an existing model. By 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.

We also focus on diminishing the effects of overfitting in this paper. When not properly accounted for, overfitting leads to failure in finance, producing false discoveries and promised outcomes that cannot be delivered [3, pp. 11–12].

This paper can contribute to the literature in two ways:

First, we used multilayer perceptron (MLP) networks in this paper. Unlike most papers on this topic that primarily focus on Long Short-term Memory (LSTM) networks or other recurrent networks [4], our approach uses a dataset we understand is unfit for LSTMs.

Our paper also differs from other MLP approaches such as [5] and [6] due to our second contribution: we structured this paper as a tutorial aimed at novice researchers.

The use of deep learning in finance is a relatively new research topic with little introductory material available for those unfamiliar with financial concepts and advanced machine learning techniques. We believe this paper could help welcome new researchers to the field by detailing each step of model development and by making our code available for public access.

By doing so, this work can also expand on [3], which introduces machine learning and financial concepts but doesn't cover deep networks.

The content of this paper is structured as follows: in section 2, the theoretical framework regarding the techniques used is presented. Section 3 explores the dataset in more detail. Section 4 details the methodology used to develop and evaluate deep learning models. In section 5, the results achieved using this methodology are presented and compared. Finally, section 6 concludes this paper with the final considerations.

2. THEORETICAL FRAMEWORK

This section briefly presents the theory behind concepts and techniques used in this paper. We believe this section is essential to the understanding of the contents in the following sections and the reasoning behind the choices in our solution. These concepts are listed in the following order: 1) neural networks and deep learning; 2) multilayer perceptron networks; 3) long short-term memory networks; 4) feature engineering and feature Selection; 5) underfitting, overfitting, and regularization; 6) cross-validation; 7) influence of random seeds on results; 8) imputation of missing values; 9) hyper-parameter tuning; 10) dimensionality reduction; and 11) ensemble, bagging, and boosting.

2.1. Neural Networks and Deep Learning

Neural networks are composed by individual 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 [7, p. 729].

Deep learning models are a type of machine learning model that take 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 [8, 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 [8, p. 35].

While neural networks have been researched for decades, deep learning only became a trend in machine learning in recent years, as neural networks 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 [8, p. 51]. [9, p. 12]

2.2. Multilayer Perceptron Networks

The most common type of deep neural networks are multilayer perceptrons, also known as deep feedforward networks. In this network layout, units are grouped into layers where each unit only receives inputs from units in the first preceding layer [9, p. 164]. After all layers executed this process, the output layer returns the predictions made and a cost function is used to evaluate how far off the predictions are compared to the true values. This error evaluation is used by the backpropagation algorithm to update the weights of all units, decreasing the total error and producing better predictions.

2.3. Long Short-term Memory Networks

LSTM networks are a type of neural network commonly used for sequential data due to their effectiveness at capturing long-term temporal dependencies [10].

They are a type of recurrent neural network that work by using feedback connections between layers. Rather than treating each input in a sequence as an independent event, an LSTM network processes entire sequences of data, retaining information about previous data in the sequence to help with the processing of new data points.

2.4. Feature Engineering and Feature Selection

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 [8, p. 151].

Feature selection, on the other hand, 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].

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.

2.5. Underfitting, Overfitting, and Regularization

A good machine learning model should be able to generalize a problem, performing well not only on the data it was trained but also on new inputs [9, p. 224].

To achieve that, 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 achieves a high training error, failing at identifying structures in the data that connect inputs to outputs and making the model perform below what would be required to solve the problem. This 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. This happens when models try to adapt to the training data in an increasingly precise way, specializing only in that data and losing generalization power.

While preventing underfitting is just a matter of developing better models, appropriate measures need to be taken to minimize overfitting. 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 [9, p. 224].

In this paper, we used dropout [11], batch normalization [12], and label smoothing [13] for regularization.

2.6. Cross-validation

Cross-validation (CV) is a validation technique widely used for monitoring and reducing overfitting. Its simplest variant, called K-fold CV, works by separating the data into k sets of similar size called folds. 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 once. At the end of the process, performance is evaluated by calculating the average of the errors found in each run [14]. Cross-validation provides a more reliable test error estimate, allowing us to see which models perform well in multiple test sets and which ones are overfitted and can only produce good results on a specific test set.

There are many other CV strategies that can be useful for different applications, such as Group K-fold, used to keep together observations pertaining to a group or category, or Time Series Split, used to separate sequential data while keeping the temporal order between individual observations [14].

Prado [3, pp. 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, 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 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, pp. 105–110].

2.7. Influence of Random Seeds on Results

Deep learning relies 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 models with significantly different accuracies [15].

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 makes certain parts of the training process to be performed with the same settings, allowing a better comparison between multiple runs [15].

2.8. Imputation of Missing Values

Many datasets contain observations with unavailable information in some of its 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 R's MICE package or scikit-learn's IterativeImputer.

2.9. Hyper-parameter Tuning

Many parameters in a machine learning model 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 hundreds of hyper-parameters. Instead, hyper-parameter tuning can be used to do this tiresome work.

The two most established methods for hyper-parameter tuning in machine learning are grid search and random search.

Grid search consists of extensively exploring all combinations of hyper-parameter values, finding those that express the best results in the entire search space. While feasible on models with a limited number of hyper-parameters, grid search is too computationally expensive for deep models with hundreds of hyper-parameters with a broad range of possible values.

Random search works by evaluating random combinations of hyper-parameter values, selecting the best-performing combination after a fixed number of random trials. Empirical evidence shows that random search outperforms grid search in complex scenarios, finding better models in less time. The reasoning for that is that hyper-parameters are not equally important to tune, and grid search allocates too many trials to the exploration of dimensions that do not matter and suffer from poor coverage in dimensions that are important [16].

Another method widely used for tuning deep networks is Bayesian optimization. Bayesian optimizers work by constructing a probabilistic model that, based on a set of initial models using random hyper-parameters, can identify which hyper-parameters contribute the most to a change in model performance. The algorithm then searches for new model configurations, with a bigger focus on parameters found to be more important in previous steps. Empirical evidence shows that Bayesian optimizers, when used to tune deep learning models trained on large datasets, outperform random search in both model performance and computational time required [17] [18].

2.10. 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 more computational resources. 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 these undesired effects. In this paper we used Principal Component Analysis (PCA) and autoencoders for dimensionality reduction.

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].

The autoencoder algorithm is another unsupervised dimensionality reduction method. It uses artificial neural networks to learn a compressed representation of its inputs.

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

When the number of units in the central layer are restricted to be smaller than the number of original input nodes, autoencoders can 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 [21].

PCA is a linear method, meaning it can't grasp the nonlinear properties present in the multidimensional data used by deep networks. It is, however, a computationally cheap technique, with fast execution even on large datasets.

Autoencoders, on the other hand, are neural networks capable of learning nonlinear relationships, with the trade-off of added computational time due to the need of training an extra neural network.

2.11. Ensemble, Bagging, and Boosting

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 [22].

One of the most used methods for ensembling models is 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].

Another commonly used ensemble method is boosting. It consists of training models in sequence rather than in parallel. For each new model trained, the findings of the previous model are taken into account, with its misclassified data having its weights increased. The next model can then focus on solving the inaccuracies of the previous, producing at the end a single predictor with a higher accuracy than all of the previous [3, pp. 99–100].

3. Dataset

The dataset provided by the competition, according to [1], 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 the following columns:

- 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. Nothing is known about what each feature and its values represent, as they are not named with descriptive labels. 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 the 130 anonymized 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. 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 evaluation dataset differs from the training dataset only by the absence of the fields *resp* and *resp_{1,2,3,4}* [1]. Access to it was not made available to participants.

4. Methodology

We adopted a quantitative and experimental approach consisting of testing and evaluating different deep learning and machine learning techniques in an incremental way. This approach is illustrated in Figure 1.

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. We opted for a fixed sequence when adding techniques to the model to reduce the complexity and the execution time of the hyper-parameter tuning step. A different order in the optimization steps could also produce good models, but testing all of them would require a lot more computational resources and extend this paper quite significantly.

The metric used to evaluate the results is the 'utility score', proposed by [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 paper was concluded after the end of the submission phase of the JSMP competition, rendering us unable to use the official evaluation from the competition to produce our results. 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 code used was written in Python, with the execution being performed using GPU acceleration. To work with 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, scikit-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. The code used in this paper is available for public access on Kaggle (https://www.kaggle.com/wendellavila/janestreet-index/) and GitHub (https://github.com/wendellavila/JSMP-Notebooks).

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 so we could group observations by date to define a fixed amount of days as the gap between train and test sets, keeping data from a same day together in each set. 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 [23, eq. (3)] was used, giving higher importance to models trained in more recent data as well as accounting

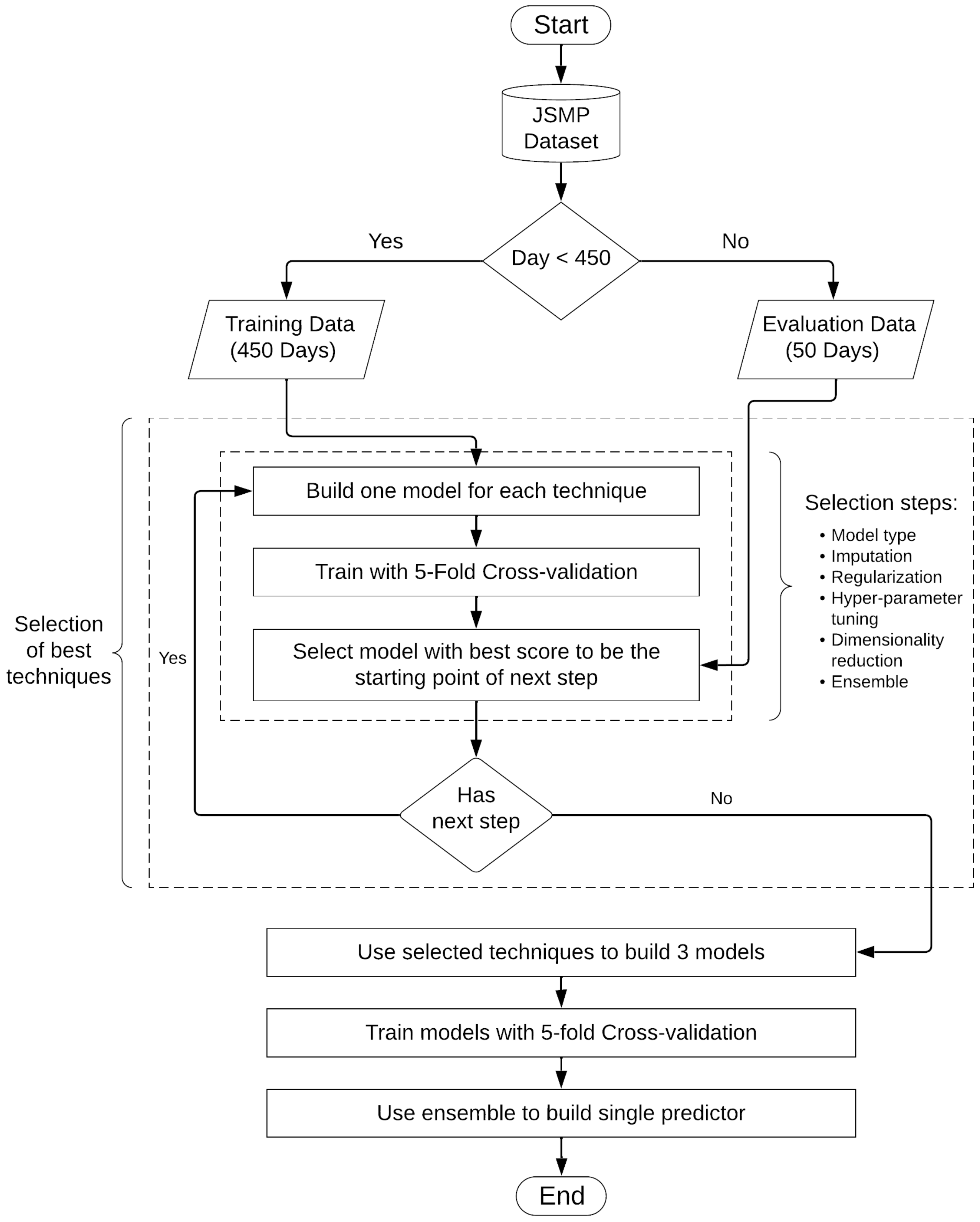
Start
JSMP Dataset
Day < 450
Yes
No
Training Data (450 Days)
Evaluation Data (50 Days)
Selection of best techniques
Build one model for each technique
Train with 5-Fold Cross-validation
Select model with best score to be the starting point of next step
Selection steps:
• Model type
• Imputation
• Regularization
• Hyper-parameter tuning
• Dimensionality reduction
• Ensemble
Yes
Has next step
No
Use selected techniques to build 3 models
Train models with 5-fold Cross-validation
Use ensemble to build single predictor
End

Figure 1: Methodology - Flowchart

for the uneven sizes of the training sets produced by the CV strategy used. To stop each training procedure, early stopping was used with the patience parameter 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 LSTM networks—widely used for sequential data—as the lack of correlation between rows in the dataset would defeat the main purpose of recurrent networks: taking previous correlated information into account when predicting new data points. Instead, MLP networks were used.

As seen in section 3, the column responsible for determining which trades are performed is not included in the training dataset. It can easily be set by applying a simple operation 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 or one. Alternatively, we could instead build a model to predict the *resp* field itself, later converting the predicted outputs into zeros or ones to get the *action* column.

Another 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.

Our initial models used the Adam optimizer and the Rectified Linear Unit (ReLU) as the activation function in layers. These are used by default in models built with Keras. A large batch size of 4096 was used to speed up training due to the large size of the dataset.

For regression, we used the mean squared error as the loss function and as the metric for early stopping. For classification models, we used binary cross-entropy as the loss function and the area under the ROC curve (AUC) for early stopping.

As stated by [9, 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 130 units, the same number as 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.

For regularization, three techniques were tested: batch normalization, dropout, and label smoothing.

To tune hyper-parameters, the bayesian optimizer Hyperopt was used. While other optimizers could be used, such as KerasTuner or Optuna, we decided to use Hyperopt for its custom objective functions. It allows us to set any custom metric to be minimized, such as the utility score described previously. 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 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. We did this to reduce the search space, allowing the search algorithm to test more values for each hyper-parameter. These steps are:

1. Tune activation functions, testing the widely used ReLU activation and the Swish activation, an activation function proposed by [24] 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. Topologies with three, four, and five layers were tested, with units in each layer ranging from 32 to 1024.
3. Tune hyper-parameters for dropout and label smoothing. Dropout rates tested range from 0.0 to 0.2 for the input layer and from 0.2 to 0.5 for hidden layers. For the smoothing label factor, a range from 0.0 to 0.5 was tested.

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

To decide the number of principal components to use as a limit to PCA, a 'scree plot' was made, suggested by [25]. This plot can help us visualize the contribution of each subsequent principal component to the summarization of the original data.

The autoencoder used is not a separate model to preprocess the data. Instead, it turns 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 [26] 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.

The diagram in Figure 2 shows how the bottleneck classifier is constructed. The autoencoder is trained before the classifier, and the weights of its encoder layer are saved in a file. This layer is then loaded and added to the MLP Classifier, forcing information to flow through a bottleneck layer to reduce its dimensionality. A Gaussian noise layer is also added to the autoencoder so it can learn how to denoise data in addition to dimensionality reduction.

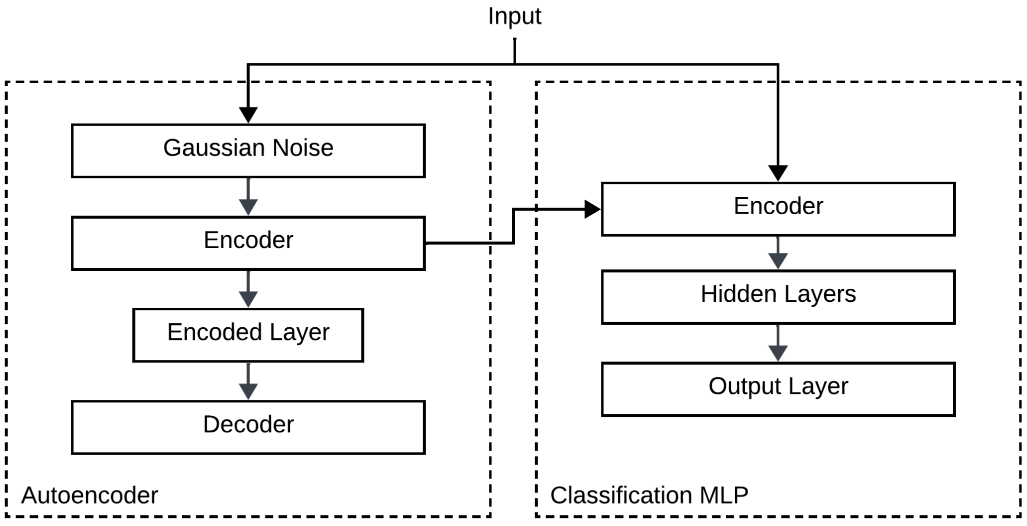

Figure 2: Deep Bottleneck Classifier

Lastly, bagging was used for ensembling models. According to [3, p. 101], it is generally preferred for financial applications, as it addresses overfitting while boosting addresses underfitting. To select more models for the ensemble, hyper-parameter tuning was performed again to select two more models, totaling 3 models with similar performance.

5. RESULTS

This section presents and compares the performance of the techniques defined in section 4 using the utility scores calculated over the validation set using the predictions of each model. The results for every model at each step of model development are listed in Table 1, while the hyper-parameters for these models are detailed in Table 2. 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. Finally, we performed the simple average of the scores achieved in these five executions to base our conclusions.

Starting with model types, results showed that the regression model had considerably worse results than classification. The multitarget approach was the one selected in this step, as it managed to achieve slightly better scores than a simple classification approach.

Following the selection of a model type, imputation methods were evaluated using the model selected in the previous step. We selected mean with missing indicator, as it showed a slight increase in the average utility score, while forward fill achieved worse results in both variants.

Regularization techniques were evaluated individually and in conjunction. In [11], dropout was used by randomly deactivating 50% of units in hidden layers and 20% of units in the input layer. For this initial analysis, we opted to use a lower rate of dropout, 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 1. Batch normalization and label smoothing, while not showing improvements as standalone regularization strategies, managed to increase dropout's scores further when coupled with it. For this reason, all three regularization techniques were used.

After incorporating regularization into the model, hyper-parameters were tuned using Hyperopt. 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.

The two objective functions used for Hyperopt 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 next step.

Starting with PCA for dimensionality reduction, the scree plot in Figure 3 showed that less than 100 principal components were enough to achieve maximum explained variance. We opted to use 80 principal components, a number significantly smaller

Table 1: Incremental model development - Results at each step

Selection step	**Model**	**Utility score (average of five executions with different seeds)**
Model Type	Regression	168.71
	Classification	581.58
	Multitarget classification	600.21
Imputation	Best of previous step + Mean	635.01
	Best of previous step + Forward fill	631.19
	Best of previous step + Mean with missing indicator	642.36
	Best of previous step + Forward fill with missing indicator	618.53
Regularization	Best of previous step + Batch normalization	564.92
	Best of previous step + Dropout	766.70
	Best of previous step + Label smoothing	558.35
	Best of previous step + Batch normalization and dropout	891.23
	Best of previous step + Batch normalization and label smoothing	545.04
	Best of previous step + Dropout and label smoothing	816.39
	Best of previous step + All 3 strategies	867.83
Hyper-parameter Tuning (New models using previously selected techniques)	Objective 1 - 3 layers	817.83
	Objective 1 - 4 layers	722.25
	Objective 1 - 5 layers	749.68
	Objective 2 - 3 layers	817.20
	Objective 2 - 4 layers	723.39
	Objective 2 - 5 layers	869.42
Dimensionality Reduction	Best of previous step + Autoencoder	876.49
	Best of previous step + PCA	792.23
Ensemble	Model 1 (Best of previous step)	811.61
	Model 2 (New Hyperopt-tuned model)	824.96
	Model 3 (New Hyperopt-tuned model)	809.62
	Bagging with models 1 and 2	829.24
	Bagging with models 1 and 3	832.04
	Bagging with models 2 and 3	843.97
	Bagging with models 1, 2, and 3	845.77

Table 2: Hyper-parameters of models used at each step

Selection step	**Model**	**Activation function**	**Units in each hidden layer**	**Dropout rate in input Layer**	**Dropout rate in each hidden layer**	**Label Smoothing factor**
Model type, imputation, and regularization	Starter Model	ReLU	130, 130, 130	0.1	0.3, 0.3, 0.3	0.1
Hyper-parameter tuning	Hyperopt objective 1 - 3 layers	swish	560, 592, 1008	0.0437	0.4622, 0.3821, 0.3044	0.1579
	Hyperopt objective 1 - 4 layers	swish	256, 288, 720, 240	0.1190	0.2368, 0.2059, 0.3489, 0.3263	0.4483
	Hyperopt objective 1 - 5 layers	swish	272, 1008, 336, 560, 672	0.1301	0.2007, 0.3748, 0.2981, 0.3524, 0.3143	0.2414
	Hyperopt objective 2 - 3 layers	swish	592, 320, 880	0.0562	0.4362, 0.3752, 0.3518	0.0704
	Hyperopt objective 2 - 4 layers	swish	496, 512, 400, 112	0.0102	0.3626, 0.2640, 0.2080, 0.2258	0.1336
	Hyperopt objective 2 - 5 layers	swish	208, 48, 112, 848, 624	0.1554	0.4770, 0.2004, 0.4212, 0.4607, 0.4318	0.4721
Dimensionality reduction	Hyperopt objective 2 - 5 layers	swish	208, 48, 112, 848, 624	0.1554	0.4770, 0.2004, 0.4212, 0.4607, 0.4318	0.4721
Ensemble	Bagging - Model 1	swish	208, 48, 112, 848, 624	0.1554	0.4770, 0.2004, 0.4212, 0.4607, 0.4318	0.4721
	Bagging - Model 2	swish	240, 352, 796, 176, 448	0.1302	0.2612, 0.4328, 0.4263, 0.4495, 0.3094	0.4746
	Bagging - Model 3	swish	416, 504, 256, 568, 368	0.1747	0.4661, 0.4207, 0.2335, 0.4132, 0.4563	0.3964

than 100 but having a similar explained variance ratio.

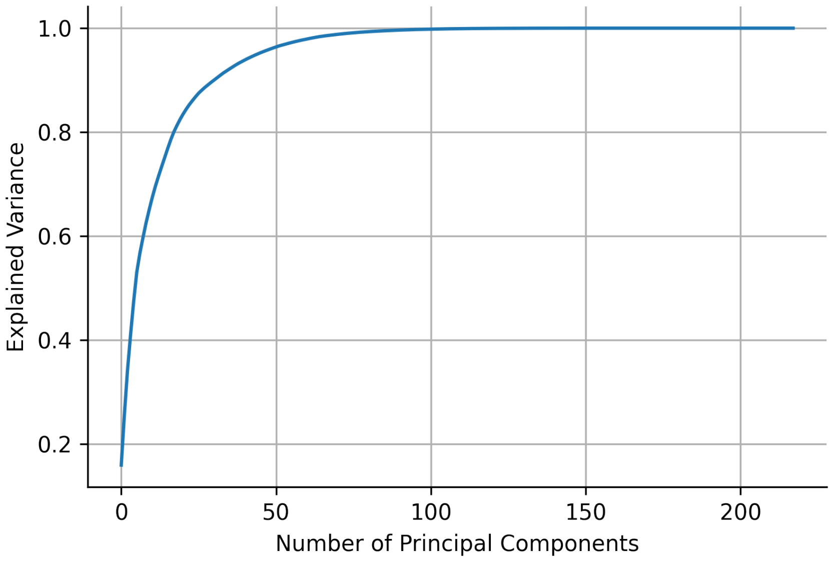

Figure 3: PCA - Scree plot

For the autoencoder, we used a traditional 3-layer topology, with the encoder layer reducing the dimensionality of the data into a single hidden layer of 64 dimensions. The Gaussian noise layer uses a standard deviation of 0.05.

The results presented led us to select the autoencoder approach for dimensionality reduction, as it managed to improve scores compared to the previous step, unlike PCA.

To perform our bagging ensemble, the objective 2 of the hyper-opt step was performed again until we managed to find two more models with similar performance of that used in the previous step.

Results showed that every ensemble of two models managed to outperform its individual models on average, with the 3-model ensemble managing to outperform any 2-model ensemble.

Finally, the models found after all these steps were combined to form a single predictor, averaging the predictions of all models trained in each of the five folds of CV over five different seeds and three different model structures. This final bagging ensemble with a total of 75 models produced a utility score of 895.62.

6. CONCLUSION

Comparing the utility score achieved by the first classification model evaluated in Table 1 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 bigger when comparing the final ensemble with the regression model, the first to be discarded, with an increase of 430.87%.

By following the proposed approach, we managed to gradually improve models, increasing the utility score at each step and achieving a final score that corresponds to 4.46% of the maximum utility score that can be achieved with 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.

6.1. Future Work

The models and techniques evaluated in our work are those we judged compatible with the JSMP dataset used and the computational resources available. Other datasets may benefit from different types of deep networks, such as LSTMs, and different training environments may allow for a more robust treatment of missing data.

Financial machine learning is not limited to deep learning, and tree-based methods such as XGBoost or LightGBM are also widely used. A similar work focused on tree-based models could also help in introducing new researchers to the field, providing a starting point for the use of these models in financial problems.

REFERENCES

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

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

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

[4] J. Huang, J. Chai and S. Cho. "Deep learning in finance and banking: A literature review and classification". *Frontiers of Business Research in China*, vol. 14, 12 2020.

[5] M. Dixon, D. Klabjan and J. H. Bang. "Classification-based Financial Markets Prediction using Deep Neural Networks". *Algorithmic Finance*, vol. 6, no. 3–4, pp. 67–77, 2017.

[6] B. X. Yong, M. R. A. Rahim and A. S. Abdullah. "A Stock Market Trading System using Deep Neural Network". In *Communications in Computer and Information Science*, volume 751, pp. 356–364, 2017.

[7] S. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, third edition, 2010.

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

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

[10] 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, no. 10, pp. 2222–2232, 2017.

[11] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014.

[12] S. Ioffe and C. Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, pp. 448–456, Lille, 2015.

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

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

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

[16] J. Bergstra and Y. Bengio. "Random Search for Hyper-Parameter Optimization". *Journal of Machine Learning Research - JMLR*, vol. 13, pp. 281–305, Feb 2012.

[17] J. Bergstra, R. Bardenet, Y. Bengio and B. Kégl. "Algorithms for Hyper-Parameter Optimization". In *Proceedings of the 24th International Conference on Neural Information Processing Systems*, NIPS'11, pp. 2546–2554, Red Hook, NY, 2011. Curran Associates Inc.

[18] J. Snoek, H. Larochelle and R. P. Adams. "Practical Bayesian Optimization of Machine Learning Algorithms". In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'12, pp. 2951–2959, Red Hook, NY, 2012. Curran Associates Inc.

[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, 2007.

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

[21] Y. Wang, H. Yao and S. Zhao. "Auto-encoder based dimensionality reduction". *Neurocomputing*, vol. 184, no. C, pp. 232–242, 2016.

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

[23] 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, 2013.

[24] P. Ramachandran, B. Zoph and Q. V. Le. "Swish: A self-gated activation function". Technical report, Google Brain, 2017.

[25] L. H. Nguyen and S. Holmes. "Ten quick tips for effective dimensionality reduction". *PLOS Computational Biology*, vol. 15, 2019.

[26] E. Parviainen. "Deep bottleneck classifiers in supervised dimension reduction". In *Proceedings of the 20th International Conference on Artificial Neural Networks: Part III*, pp. 1–10, 2010.