APPLYING RECURRENT NEURAL NETWORKS TO PREDICT THE BANKRUPTCY OF CORPORATIONS

May 22nd, 2023

IE Business School

Master in Finance

Junze Shen
Zhe Wu
Supervisor: Antonio Rivela Rodriguez

Abstract

This academic paper scrupulously explores the feasibility of advanced deep learning algorithms, particularly the Recurrent Neural Networks (RNNs), for the prognostication of corporate defaults. The primary function of these neural architectures is to formulate a prediction concerning the prospect of a corporation's bankruptcy within one year. This prediction leverages a five-year historical financial data of approximately 40,000 companies, their market performance metrics, and macroeconomic indicators. The fundamental ambition of this investigation is to ascertain the optimal configuration for such a task and to critically evaluate the suitability of RNNs for such a challenging application.

Table of Contents

1. Introduction

1.1. Objective

This scholarly composition aims to meticulously evaluate the efficacy of Recurrent Neural Networks (RNNs) in predicting the likelihood that a company will default within the span of a year with a commendable degree of precision. The challenge of forecasting a company's bankruptcy is a critical issue in finance and the pursuit of an effective solution is crucial in all types of financial decision-making processes.

Our objective is to ascertain whether Recurrent Neural Networks (RNNs) serve as a suitable methodology for this issue. To achieve this, it is imperative that we construct, test, and subsequently refine an RNN architecture to generate the highest accuracy with the lowest bias for each class.

If our outcomes are positive and trustworthy, we still need to run thorough and rigorous tests before we make an unequivocal affirmation that an RNN deep learning model has higher superiority over the capability of humans at answering this question of prognostication of a corporation's default. Moreover, the need still prevails to determine that the employment of such a profound and costly learning model is, indeed, economically justifiable in fulfilling the targeted objective.

1.2. Methodology

The methodology employed in this paper encompasses three principal aspects: the compilation of the dataset, the construction of a conceptual model, and the derivation of predictive models.

The process of training and evaluating a deep learning model necessitates an extensive amount of data. The invaluable data we utilize for this purpose is graciously obtained from Professor Antonio Rivela Rodriguez. This substantial trove of information has been mined by his exceptional previous students, Juan Pablo Bonfrisco Ayala and Javier de Vicente Moreno, from the Bloomberg database. As scholars Junze and Zhe, we fully take advantage of using this carefully curated dataset and implement certain modifications to the data. The specific details of these adjustments will be explained in the subsequent sections of this document.

Throughout the process of creating a conceptual model, we have enjoyed the experience of perusing and understanding a wealth of informative academic publications by pioneering figures like Bai, Yuhan, and Bengio. Our methodology is to assimilate and comprehend the critical components of neural networks, including diverse layers, activation functions, regularization techniques, and a set of hyperparameters. We believe that such an investment of time and effort is instructive in the creation of a coherent and justifiable conceptual model that aligns with our objectives. As we traverse this process, we obtained a profound comprehension of a Recurrent Neural Network (RNN) model and understood its efficacy in predicting corporate defaults.

In pursuit of the optimal model that would best meet our objectives, we created five candidate models and engaged in over 1,000 iterations, designed to reduce the models' uncertainties. Although such an endeavor is costly and time-consuming, we assert that this level of commitment was a necessary sacrifice to achieve our goal, which is not trivial and superficial to humanity. Our confidence in the predictive models is cemented by the results derived from Monte Carlo Simulations. It is our aspiration that our diligence will yield meaningful results and contribute to the application of deep learning models for the financial market.

2. Data Description

2.1. Basic Information

The dataset employed in this paper is credited to previous efforts of students working in the same subject, which is time series data consisting of 28 attributes (Financial, Market, and Stock), 5 consecutive years of 39,536 companies[1] (396 bankrupt and 39140 healthy) data sourcing from Bloomberg. A sample data of bankrupt company is presented in Table 1 shown below:

Attributes	**Years**				
	Year 1	**Year 2**	**Year 3**	**Year 4**	**Year 5**
YoY % in stock price	-0.42982456	-0.07692308	1.38333333	0.16083916	-0.8554216
market cap Jun's data	187.2812	172.29	414.2352	609.9877	87.756
90-days stock vol	0.46791	0.48373	0.60221	0.39215	1.05035
…	…	…	…	…	…
10-year risk free	0.0708	0.0658	0.06438334	0.063525	0.05264167

Table 1: sample data illustration of bankrupt company NO.1

Note that all the companies follow the same order of attribution as it is shown in the above Table. Furthermore, missing data in the dataset, such as bank sector corporations do not typically have inventory, is treated as 0 to keep the data realistic.

2.2. Data Preprocessing and Cleaning

It is essential to perform data preprocessing and cleaning since the raw data provided does not satisfy the input data requirements for the Keras deep learning RNN models. To build and train the model properly, it is necessary to perform: transpose and split, label and train-test-split, outlier removal and normalization, and data balancing. The following subsections provide a detailed description as well as the rationale behind each procedure.

2.2.1. Transpose and Split

The data given consist of 2 separate Excel files each corresponding to data from bankrupt companies, and data from healthy companies. With those two-dimensional data with shapes (11088, 5) and (1095920, 5) correspondingly as well as the knowledge of the total NO. of bankrupt and health companies, we first splitted each dataset by 28 attributes and obtained a three-dimensional dataset with shape (396, 28, 5) and (39140, 28, 5) with 396 and 39140 being the number of companies and (28, 5) being one block consisting 5-years' 28 attributes. Then, in order to meet the input shape requirement of RNN architecture presented by Keras library which states that the input shape of a time series data for RNN should be (batch, time, attribution), we

[1] In the original dataset provided, for companies for which have more than five years of the attributes, we consider each subset of five consecutive years of attributes as one member of the population. Thus, strictly speaking the total NO. of different firms is less than 39536.

transposed the dataset into (396, 5, 28) and (39140, 5, 28) respectively. The code performing this task is provided below:

```
bc_data2 = np.array(np.split(bc_data1.T,396,axis=1))
hc_data4 = np.array(np.split(hc_data3.T,39140,axis=1))
```

2.2.2. Label and Train-test-split

As it is stated previously, the methodology used in this paper is time series binary classification with supervised deep learning. Thus, to provide the correct and proper dataset to our model, it is necessary to label the health and bankrupt companies. In our case, we defined the healthy companies as 0 and bankrupt companies as 1 and assigned the labels to each block of data in each group. Subsequently, with our belief that it is unreasonable to remove the outlier for test and validation datasets since otherwise, it will affect the degree of realisticity, we first randomly chose 100 blocks from the bankrupt companies plus 100 blocks from the healthy companies as our test dataset and 76 blocks from bankrupt companies plus 76 blocks from the healthy companies as our validation dataset and then set the remaining blocks as our training datasets, which was subject to outlier remover in the next step.

2.2.3. Normalization and Outlier Remove

Data normalization is a crucial step in the preprocessing stage of deep learning. It involves adjusting the values measured on different scales to a common scale because of reasons, including speed-up training, avoiding gradient issues, handling outliers, and consistency. When attributes are on the same scale, the backpropagation algorithm takes less time to find the optimal weights, making the whole learning process more efficient. Moreover, some attributes might have a significantly larger range of values than others, such as market capitalization in million and risk-free rate in small percentage numbers in our datasets. Such large values can dominate the direction of the gradient during training, leading to overshooting the minimum or divergence from the goal. Additionally, by doing normalization, it can be easily done to remove outliers using a predefined range. For the purpose of consistency, it is extremely difficult for the model to learn effectively from the data without normalization. Hence, normalization is an indispensable step in this project, which makes everything easier to process.

In our research, the data normalization and outlier remove tasks are performed by the following code:

```
def outlierremover(data,threshold):
  mu = np.mean(data.T.reshape(data.shape[2],data.shape[0]*5),axis= 1)
  std = np.std(data.T.reshape(data.shape[2],data.shape[0]*5),axis= 1)
  z = (data - mu) / std
  mask = (z >= - threshold) & (z <= threshold)
  clean_data = data[np.all(mask, axis = (1,2))]
  return clean_data


mu_train =
np.mean(train_x.T.reshape(train_x.shape[2],train_x.shape[0]*5),axis=
1)
std_train =
np.std(train_x.T.reshape(train_x.shape[2],train_x.shape[0]*5),axis=
1)
train_x = (train_x - mu_train) / std_train
test_x = (test_x - mu_train) / std_train
val_x = (val_x - mu_train) / std_train
```

As it is present above, the two inputs are data needed to be processed and threshold, which is the predefined range (i.e.how many standard deviations away from the mean) that we wanted our data to be contained. The mathematics behind normalization is trivial, which only requires computing the mean and standard deviation respectively for each attribute (28 in total). Then calculate the Z-score for the data provided, which needs it minus the mean and divided by the standard deviation. After normalization, the training dataset has a mean of zero and a standard deviation of one The bottom two lines of codes above indicate that the entire 5 by 28 data for the company is removed from the dataset if one of the Z-scores of that company exceeds the threshold. In other words, we reserved data for those companies where all the Z-scores lie within the range. The most notable thing is that we only calculate the mean and standard deviation for the training dataset and apply those two figures to our validation and testing dataset to avoid "data leakage".

The code below shows what we've actually done to our dataset.

```
bc_new = outlierremover(bc_data2[n_test_bc+n_val_bc:],threshold = 2000)
hc_new = outlierremover(hc_data4[n_test_bc+n_val_bc:],threshold = 4.5)
```

We set the threshold of bankrupt companies' dataset to 2,000. Such a high threshold indicates that we didn't remove any data from the bankrupt dataset. This is due to the fact that we want to keep the level of reality in our bankruptcy dataset and we don't want to lose any information. On the other hand, we set the threshold of 4.5 for the healthy dataset, which consists of 39,140 healthy companies in total. After the outlier remover function, we keep companies that have all the data within 4.5 standard deviations from the mean and 36,553 healthy companies left.

2.2.4. Data Balancing

Since our original dataset is extremely imbalanced, with approximately 99% of healthy companies and 1% bankrupt companies, it is necessary to carefully balance the data because class imbalance can dramatically affect the performance of the model in binary classification tasks.

First of all, our model aims to maximize the overall accuracy of classification. If we did not balance our training dataset, this would completely fail to identify the minority class, which is the class of interest (bankrupt companies) in this project. The model would have high accuracy overall, but the model would be super biased towards the majority class and overfit to healthy companies.

The most commonly used balancing strategies include: undersampling, oversampling and weighted classes. We finally chose to use weighted classes instead of resampling based on the following rationales:

A. Using undersampling would lose too much information because the healthy companies are much more than bankrupt companies and this would increase variance by a lot. The result might be incorrectly represented.

B. Using oversampling might create overfitting because of duplicating and misleading instances like SMOTE (Synthetic Minority Oversampling Technique), which might not always represent plausible real-world cases.

C. Our dataset has many samples with 5 time series and 28 different feature data, which could be considered high-dimensional data. Using synthetic examples might distort the feature space and make the decision boundaries less clear and the technique is ineffective due to high dimensionality, learning to less meaningful synthetic.

Consequently, we decided to utilize the class weights approach because it is flexible to use (adjust the weights) and saves many computational costs. In our case, the class weights method was performed by the code present below:

```
class_weight_auto  = class_weight.compute_class_weight(class_weight =
"balanced",classes = np.unique(train_y),y = train_y)
```

The "compute_class_weight" function imported from sklearn.utils library computes the class weights given the data and corresponding labels. In this case, balanced class weight was used, which means that the function will use the values to automatically adjust weights inversely proportional to class frequencies in the input data. In our case, the output should be a dictionary of weights: **{0: 0.50, 1: 84.20}** where we attributed the 0.50 weight to healthy companies and 84.20 weights to bankrupt companies, which could be used to weigh the loss function introduced later in this paper.

Note that it is nonnegligent to be aware of the potential drawbacks of using class weight. For example, it still has the possibility of overfitting and it is difficult to find the optimal weights between classes.

2.3. Final Data Summary

After the entire data preprocessing we introduced above, the final dataset can be described as the following characteristics:

A. Training dataset consists of shuffled 36,773 companies (36,553 healthy and 220 bankrupt). The sample is normalized with a mean of zero and a standard deviation of one.

B. Validation dataset consists of shuffled 76 healthy companies and 76 bankrupt companies. Normalized using the mean and standard deviation of the training dataset, not the validation dataset.

C. Testing dataset consists of shuffled 100 healthy companies and 100 bankrupt companies. Normalized using the mean and standard deviation of the training dataset, not testing dataset.

Sample illustration of a single block of training dataset is presented in Table 2:

Years	**28 Attributes**					
	Δ Stock price	Market cap	Stock Vol	Trading volu		Risk-free
Year 1	-0.419	0.673	2.953	0.238	…	0.064

Year 2	0.444	0.666	3.340	0.506	…	0.053
Year 3	0.231	1.196	4.143	0.231	…	0.056
Year 4	4.125	10.86	0.853	11.79	…	0.060
Year 5	-0.863	1.672	1.471	0.850	…	0.050

Table 2: sample data after preprocessing illustration of bankrupt company NO.1

3. Conceptual Model

3.1. Introduction to Conceptual Model

In the context of deep learning, a conceptual model consists of abstract notions aimed to design and construct a model based on logical reasoning. This model aims to comprehend, explain and simulate the most important issues under consideration. The conceptual model is a necessary step in the process of building a prediction model, but it's not the final product.

3.2. Model Architecture

3.2.1. The Choice of RNN Layer (LSTM)

Since the project is dealing with a five-year time series data with twenty-eight different attributes, it is optimal to use Recurrent Neural Networks. This is due to the fact that RNNs have the ability to maintain hidden states that allow them to incorporate historical patterns from the data, which establishes temporal dependencies between different points in the time series. The most usually practiced RNN layers are SimpleRNN, LSTM, and GRU. However, standard RNNs can struggle to carry information across many time steps due to what is known as the "vanishing gradient" problem, which makes it hard for the RNN to learn and maintain time dependencies in the data.

The structure we decided to use is LSTM (Long Short-Term Memory) since it addresses the issue of “vanishing gradient” with a special architectural feature called gates. Besides, LSTM maintains a “cell state” that gets updated over time.

The LSTM layer traditionally employs the hyperbolic tangent (tanh) and the sigmoid activation functions in their internal workings and it is proven that the use of these functions is not arbitrary but is linked to the design and theoretical underpinnings of LSTM units.

The brief properties and identities of Sigmoid and Hyperbolic Tangent functions are illustrated below:

The sigmoid function is a special form of the logistic function and is usually denoted by σ(x) or sig(x). The equation is presented below and graphs of sigmoid function are presented in the Appendix (Figure 1 & 2):

$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad \sigma'(x) = \sigma(x)(1-\sigma(x))$$

This function maps any real number to a range (0,1) not included boundaries. This means that the output of the sigmoid function can be interpreted as a probability, making it especially useful in models that estimate binary outcomes (like logistic regression).

Hyperbolic tangent Activation Function, tanh, is also like logistic sigmoid but better to deal with negative values because the output range of the tanh function is from (-1 to 1).

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \qquad tanh'(x) = 1 - tanh(x)^2$$

Similar to the sigmoid function, tanh is also S-shaped or sigmoidal in form, but it is symmetrical around the origin since the output is zero-centered.

Both sigmoid and hyperbolic tangent activation functions suffer from the problem of "vanishing gradients". When the absolute value of the input is large, tanh produces outputs very close to -1 or 1, where the gradients (first derivative) are close to zero.

The mechanism of the specialty of the LSTM layer is demonstrated in the following section:

In the mathematical context, consider at each time step t, we have input $x[t]$, previous hidden state h[t-1], and cell state c[t-1].

1) **Forget gate**: determines which information to be forgotten in the cell state

$$f[t] = sigmoid(W_f \cdot [h[t-1], x[t]] + b_f)$$

Where the concatenated matrix h[t-1] and x[t] is multiplied by the forget matrix W_f and b_f is the bias vector. Then the value is the input of a sigmoid activation function inside the LSTM layer.

2) **Input Gate**: updates the cell state with new information

$$i[t] = sigmoid(W_i \cdot [h[t-1], x[t]] + b_i)$$

$$C[t] = tanh(W_C \cdot [h[t-1], x[t]] + b_C)$$

It has two parts: a sigmoid layer called the "input gate layer" and a tanh layer that creates new candidate values that could be added to the state.

3) **Cell State Update:** update the old cell state c[t-1] into the new cell state c[t]

$$c[t] = f[t] \cdot c[t-1] + i[t] \cdot C[t]$$

Note that c[t] represents the cell state at time t, while C[t] represents the new candidate value.

4) **Output Gate:** based on the cell state

$$o[t] = sigmoid(W_o \cdot [h[t-1], x[t]] + b_o)$$

$$h[t] = o[t] \cdot tanh(c[t])$$

As mentioned previously, the use of tanh here ensures the LSTM's output is normalized and can smoothly vary between -1 and 1.

In conclusion, the LSTM function allows the model to handle negative and positive values evenly and normalize outputs. This is crucial because our input data has already been normalized with mean zero and standard deviation one. Using other activation functions may lose some input information.

3.2.2. The Number of Layers

We chose to reduce the complexity of our logical model because the input data shape is (5,28), which indicates a relatively simple input source for a deep learning algorithm. Accordingly, we only built three layers for our logical model, consisting of an LSTM layer at the top, followed by a dense layer, and concluded by another dense layer on a sequential basis. The advantages of using simple architecture are manifold, such as avoiding overfitting, computational efficiency, and interpretability.

3.2.3. The Number of Perceptrons

We hold our opinion that determining the number of perceptrons (neurons) in a neural network is more of an art than a science. Based on our purpose of simplicity and efficiency, we arbitrarily determined the number of neurons to be the power of 2. As a result, the LSTM layer has 64 neurons, the first dense layer has 8 neurons and the output dense layer has 1 neuron. Since this decision is more of an art, finding the best number of neurons usually involves a mix of experience, intuition, and trial and error.

3.3. Activation Functions

3.3.1. Introduction to Activation Function

In deep learning, the choice of activation functions for each layer is vital. An activation function is a mathematical function that maps a neuron's input signal into an output signal in the setting of neural networks. This output signal is subsequently fed into the network's next layer. Mathematically speaking, the internal mechanism of transformation between layers can be categorized as the following equation:

$$Next = \sigma\left(\begin{bmatrix} w00 & w01 & w02 & ... & w0n \\ w10 & w11 & w12 & ... & w1n \\ . & . & . & . & . \\ . & . & . & . & . \\ wk0 & wk1 & wk2 & ... & wkn \end{bmatrix}\begin{bmatrix} a0 \\ a1 \\ a2 \\ . \\ . \\ an \end{bmatrix} + \begin{bmatrix} b0 \\ b1 \\ b2 \\ . \\ . \\ bk \end{bmatrix}\right)$$

where sigma is the activation function, “w” is the weights assigned to each input, “a” is input, “b” is the bias.

In summary, the goal of activation functions is to introduce nonlinearity into each transition among neurons, allowing the neural network to learn from erroneous backpropagation and model complicated interactions between its inputs and outputs.

3.3.2. Activation Function Choice

As it is previously mentioned plus our experiment, the LTSM layer in Keras only accepts sigmoid and tanh as activation functions. Though it will still function if other activation functions are given, a warning sign will appear indicating that only tanh and sigmoid will be employed. Thus, for our LSTM layer, we decided to be aligned with the initial setup and chose sigmoid and tanh as activation functions.

In terms of our determination of the activation function for the second Dense layer, we chose PReLU (Parametric Rectified Linear Unit). To demonstrate our rationale behind this choice, it is intuitive to present the limitation of choosing other common activation functions as well as the advantage of choosing PReLU in our circumstances. Note that all the graphs are presented under Appendix (Figure 3-6).

1) ReLU Function:

$$f(x_i) = \begin{cases} x_i, & \text{if } x_i > 0 \\ 0, & \text{otherwise} \end{cases}$$

Although it is previously demonstrated that ReLu is simple and efficient in the backward propagation process and can prevent potential Vanishing Gradient problems, it is unreasonable to employ ReLU in our model since it will generate "Dying relu" problems. Namely, as the previous LSTM layer employs tanh function to transfer its outputs, the output values will all possess the range from -1 to 1. And since the ReLU function will only consider the positive value, the information contained in the negative values will be omitted, resulting in inaccurate interpretation.

2) LeakyReLU:

$$f(x_i) = \begin{cases} x_i, & \text{if } x_i > 0 \\ \alpha_i x_i, & \text{otherwise} \end{cases}$$

Where: α_i is typically between 0.01 and 0.2

In the past few years, there have been many efforts to solve the "dying relu" problem and LeakyReLU is one of them. LeakyReLU solves part of the problem by giving a small weight to the negative value, which we considered inappropriate for our model also since it possesses a clear bias towards positive values. Additionally, Dr. Yuhan Bai, in his research paper, also indicated that the LeakyReLU also "cannot avoid the problem of gradient explosion, and the neural network cannot learn the alpha value."[2]

3) ELU & PReLU:

$$f(x_i) = \begin{cases} x_i, & \text{if } x_i > 0 \\ \alpha_i(e^{x_i} - 1)), & \text{otherwise} \end{cases} \qquad f(x_i) = \begin{cases} x_i, & \text{if } x_i > 0 \\ \alpha_i x_i, & \text{otherwise} \end{cases}$$

According to the research papers published in 2015[3] and 2016[4], both activation functions are superior in performance and efficiency than ReLu or LeakyReLU. However, each of these functions possesses its own drawbacks and thus makes it difficult to compare the performance in regards to the superiority of these two functions. In our model, we chose to employ PReLU instead of ELU in view of the fact that ELU will saturate the larger negative values and also ELU is less computationally efficient since it introduces the exponential factor.

[2] Bai, Yuhan. (2022). RELU-Function and Derived Function Review. SHS Web of Conferences. 144. 02006. 10.1051/shsconf/202214402006.

[3] He, Kaiming & Zhang, Xiangyu & Ren, Shaoqing & Sun, Jian. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. IEEE International Conference on Computer Vision (ICCV 2015). 1502. 10.1109/ICCV.2015.123.

[4] Clevert, Djork-Arné & Unterthiner, Thomas & Hochreiter, Sepp. (2015). Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). Under Review of ICLR2016 (1997).

Lastly, for the choice of the final (output) Dense layer, it is trivial and reasonable to choose sigmoid as our primary objective is binary classification.

3.4. Loss Functions

The essence of a deep learning model is the choice of the loss function and how to minimize this loss function. A loss function in a neural network is a measure of how well the network is performing its task. It calculates the difference between the network's prediction and the actual target value for a given input.

Since the ultimate goal of this deep learning model is to predict the default of companies using five years of financial data, it should be considered a binary classification problem because there are only two possible outcomes in this model, either default or not. The best choice of loss function according to our logical model is ***Binary Cross-Entropy*** because it provides a probabilistic interpretation of the result to measure the model's confidence in its predictions. In addition, Binary Cross-Entropy penalizes false positives and false negatives relatively equally, which is indeed an important property when both types of errors are equally undesirable. The reason that our final activation function is the sigmoid, which is a natural combination as the sigmoid function outputs a value between 0 and 1. The Binary Cross-Entropy loss function then compares this predicted probability with the true label. According to previous research[5], the mathematical inside the loss function is explained below:

$$\mathcal{L}(y,p) = \sum_{i=1}^{d} y\log(p) + (1-y)\log(1-p)$$

where y is the actual label and p is the predicted value by the model. The first term calculates the loss if the true label y is 1, while the second term calculates the loss value if the true label y is 0. This equation is symmetrical for both label zero and one, which outputs a loss value with no clear bias to either class. It's worth noting that p should ideally not be exactly 0 or 1 to avoid undefined values when calculating the log in the formula. However, this is automatically ensured by the sigmoid activation function incorporated, which outputs a value from 0 to 1 exclusively.

3.5. Optimizer

The optimizer function in a deep learning model also contributes fundamentally to the general performance of the model. It changes the attributes of parameters such as weights and learning rate in order to reduce the losses given by the loss function. Optimizers help to minimize the loss faster and more accurately. A good optimizer will adjust the weights and biases to reduce loss in the most efficient way possible, enabling the neural network to learn as effectively as possible. Therefore, a good choice of optimizer would improve the general success of the entire model.

The common practice of optimizers includes Stochastic Gradient Descent (SGD), Adam (Adaptive Moment Estimation), and RMSprop (Root Mean Square Propagation). Among these, we finally used ***Adam*** as the optimizer based on the original paper[6].

The rationale behind Adam Optimizer could be briefly explained by the following steps: consider the parameters t: time step, which is also the number of iterations; θ: parameter factor;

[5] Ruby, Usha & Yendapalli, Vamsidhar. (2020). Binary cross entropy with deep learning technique for Image classification. International Journal of Advanced Trends in Computer Science and Engineering. 9. 10.30534/ijatcse/2020/175942020.

[6] Kingma, Diederik P. and Jimmy Ba. "Adam: A Method for Stochastic Optimization." CoRR abs/1412.6980 (2014): n. pag.

α : learning rate; β_1, β_2: exponential decay rates (close to 1); ε: small constant for numerical stability; m, v: first and second-moment vector, mean and variance respectively, initialized as 0.

Subsequently, for each iteration:

A. Compute the gradient for the loss function: $g_t = \nabla_\theta \cdot f_t(\theta)$

B. Update the biased mean estimate: $m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$

C. Update the biased variance estimate: $v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$

D. Correct the bias for the mean: $\hat{m} = m_t / (1 - \beta_1^t)$

E. Correct the bias for the variance: $\hat{v} = v_t / (1 - \beta_2^t)$

F. When t approaches infinity, both $1 - \beta_1^t$ and $1 - \beta_2^t$ tend to be 1.

G. Update the parameters: $\theta = \theta - (\alpha \cdot \hat{m_t}) / (\sqrt{\hat{v_t}} + \varepsilon)$

H. Repeat step 1-7 until the algorithm converges to a solution.

One of the key advantages of Adam is that it employs the estimates of the first and second moments, which is the mean and variance of the gradients to adaptively change the learning rate for different parameters.

Accordingly, the Adam function only computes the first-order gradient, which makes it more computationally efficient than other optimization functions which require higher-order derivatives. This characteristic causes Adam to deal with high dimensional parameters like the input data in this project. Moreover, Adam is relatively less sensitive to the initial learning rate and hyperparameters, unlike SGD whose learning rate must be well-tuned.

3.6. Regularization

In both deep learning and machine learning, the problem of overfitting has long been discussed and concerned since it will severely limit the prediction power of the model. In short, overfitting occurs when the model gets so complex in terms of parameters that it not only just learns the underlying patterns, but also the random noises and outliers in the training dataset, resulting in a low error in the training dataset but a fairly high error when implementing the test dataset. A graphic representation of overfitting is presented in Appendix.

In neural networks, the major parameter to be regularized is the weight matrix and three major regularization methods, L1, L2, and Dropout, are introduced in this paper to prevent potential overfitting and Dropout method is chosen in our model due to the presence of co-adaptation. The detailed reason behind this choice, as well as the explanation of each method, are presented in the Appendix (Figure 7):

1) L1 Regularization (Lasso Regression)

   L1 regularization method performs the task by introducing a penalty term in the loss function:

$$Loss = BinaryCrossEntropy(y, p) + \lambda \sum_{1}^{n} |w_i|$$

where the λ refers to the regularization rate and w_i refers to the weights involved in the network.

By observing the gradient of the L1 loss function presented below, it is clear that L1 regularization will decrease the value of high-valued weights for each backpropagation iteration step.

$$w_{\text{new}} = \begin{cases} w - \eta \cdot [2x(wx + b - y) + \lambda] & w > 0 \\ w - \eta \cdot [2x(wx + b - y) - \lambda] & w < 0 \end{cases}$$

Moreover, one of the crucial characteristics if the weight is greater than 0, a regularization rate λ will be subtracted from the old weight; and if the weight is less than 0, the rate will be added to the old weight. Consequently, L1 regularization tends to make the overvalued weights towards 0 and is suitable for the purpose of feature selection. However, with our initial intention to employ the whole attributes provided, it is unreasonable for us to omit some of the attributes in the regulation process. Thus, we chose not to employ the L1 method.

2) L2 Regularization (Ridge Regression)

Similar to L1 regularization, L2 regularization possesses the functional form of:

$$Loss = BinaryCrossEntropy(y, p) + \lambda \sum_{1}^{n} w_i^2$$

Different from the penalty terms in L1 which adds the sum of, L2 adds the sum square of the weights, which denotes that instead of pushing weights directly towards 0, L2 regularization tends to penalize the weights to small values, but not necessarily 0. A detailed equation of the gradient of L2 loss function is presented below.

$$w_{\text{new}} = w - \eta \cdot [2x(wx + b - y) + 2\lambda w]$$

One of the latent characteristics of L2 regularization is its ability to address the potential multicollinearity in the model. However, according to previous research[7], neural networks, in general, are less likely to suffer from multicollinearity due to the overparameterization. Thus, we decided not to employ the L2 regularization technique.

3) Dropout Layer

Unlike L1 and L2 regularization, dropout is a relatively new technique specifically employed in deep learning. Dropout refers to the removal of neurons in the hidden layer nodes from a neural network. When a node is dropped, all the connections associated with it, both forward and backward, are temporarily eliminated, resulting in a tweaked network structure derived from the original network with the probability of dropping a node denoted by “p”. Such a technique is especially effective in neural network architectures since it solves the problem of co-adaptation. Co-adaptation is a recurrent problem in neural networks which states that many nodes or parameters within the network become unduly reliant on one another to execute a task. It happens when particular nodes or parameters

[7] De, Richard & Lyle, Veaux & Ungar, H.. (2000). Multicollinearity: A tale of two nonparametric regressions. 10.1007/978-1-4612-2660-4_40.

establish substantial interdependencies during the learning process, relying heavily on each other's outputs to function properly. Thus, we chose the Dropout approach to prevent potential overfitting.

3.7. Other Hyper-Parameters

Different deep learning algorithms involve different sets of hyper-parameters. For our interest, we focused mostly on the ***batch-size*** and number of ***epochs.***

3.7.1. Batch-Size

In the context of deep learning neural networks, batch size refers to the number of training examples used in one iteration. The type of batch size plays a momentous role in the training process and can affect both the accuracy of the model and the speed of training. There are four categories of batch size used by practitioners: Batch Gradient Descent (i.e. batch size = training size), Stochastic Gradient Descent (i.e. batch size = 1), Mini-Batch Gradient Descent (i.e. 1 <= batch size < training size), and Online Batch (i.e. variable size).

We chose to employ the mini-batch size method due to the fact that it is the most common approach in practice and can provide a good balance between speed, memory usage, and gradient accuracy. According to the research paper written by Yoshua Bengio[8], the number of mini-batch sizes should be the power of 2 and a batch size of 32 is a good default value due to reasons related to the architecture and performance optimization of modern GPUs. Theoretically, it influences the training time more than the test performance, meaning the general performance is less likely to be affected by the batch size. Consequently, after our evaluation and fine-tuning, we decided to use a batch size of **1,024** in the architecture primarily due to the reason that the training dataset is relatively huge and it is beneficial to save computational time.

3.7.2. Number of Epochs

In the context of training a neural network, an ***Epoch*** refers to one complete pass through the entire training dataset during the training process. Within each epoch, the model's parameters are updated in one attempt to minimize the loss function. The number of epochs determines how many times the learning algorithm will work through the entire training dataset. It is crucial to select the appropriate number of epochs to avoid underfitting or overfitting to the testing dataset. Based on the same paper written by Yoshua Bengio, this hyper-parameter is fairly easily tuned by a technique called "early stopping". Early stopping refers to the concept of tracking the out-of-sample error (or loss) as training progress and thus is able to stop the training iteration process when the out-of-sample errors or loss are continuously raising for the validation dataset. Based on such a fact, in our paper, we took advantage of using the "callback function" in Keras library to monitor the validation loss based on the principle of early stopping in order to further prevent strong overfitting by choosing the number of epochs 40 and callback step of 3. More specifically, as long as the callback function detects a 3 consecutive increments in validation loss during the training iteration process, it will automatically stop the iteration and complete the training.

3.8. Conceptual Model Summary

Based on the information described above, the Python code building this conceptual model is presented below and the graph representation of the architecture is presented under Appendix (Figure 8):

[8] Bengio, Yoshua. (2012). Practical recommendations for gradient-based training of deep architectures. Arxiv.

```
model = keras.Sequential()
model.add(keras.layers.LSTM(64, input_shape=(5,28),
activation='tanh',dropout = 0.1))
model.add(keras.layers.Dense(8, activation='PReLU'))
model.add(keras.layers.Dropout(0.125))
model.add(keras.layers.Dense(1, activation='sigmoid'))
model.compile(loss= tf.keras.losses.BinaryCrossentropy(), optimizer =
tf.keras.optimizers.Adam(), metrics = METRICS)
es_callback = keras.callbacks.EarlyStopping(monitor='val_loss',
patience=3)
model.fit(train_x, train_y, validation_data=(val_x, val_y),epochs = 40,
batch_size = 1024, callbacks =
[es_callback],class_weight=class_weight_auto,verbose = 0)
```

Subsequently, we trained and tested this conceptual model by iterating 1000 times with each time a different test, validation, and training dataset. The detailed performance of this model is summarized in Table 3:

Models	**Performance attributes**			
	Mean Accuracy	Max Accuracy	Min Accuracy	Std of Accuracy
LSTM Model	81.45%	89.90%	74.00%	0.0297

Table 3:performance summary of conceptual model

4. Prediction Model

4.1. Introduction to Prediction Model

In contrast to a conceptual model which is based on logic and reasoning to understand a problem to construct a neural network, a prediction model is selected, from various conceptual models, based solely on the accuracy and prediction power, even if the model is counterintuitive. In other words, a prediction model is the one that possesses the best accuracy on a test dataset among other conceptual models. The predictive model has learned the patterns from past data during the training phase and uses them to classify new incoming data for practical purposes. In a word, prediction models are a cornerstone of the utility of neural networks.

In theory, two of the most common approach to select proper prediction models from conceptual models are: one is by fixing the training dataset and fine-tuning both parameters and hyper-parameters by iteration based on a single conceptual model until it reaches the target accuracy; the other is by building several conceptual models using similar parameters and hyper-parameters and training those models by iterating through different, randomly chosen training datasets, within which the models get trained to the same dataset in each iteration. The graphic representations of both approaches are presented in Appendix.

For the purpose of realisticity, we decided to employ the second method to select a prediction model.

4.2. Candidate Models and Methodology

In this paper, other than the LSTM model we had built previously, other four conceptual models with different architectures were built and they were treated as potential candidates for the final prediction model. The code of each of those five models is presented in Appendix.

Since our methodology involved in choosing the testing dataset randomly, the performance of models each time had a high degree of uncertainty and variance. Hence, the result would be meaningless if we only run the models once. In order to make the performance comparison among models reasonable and logical, we trained all five candidate models a thousand times with the same randomly chosen dataset each time for each model. Subsequently, we computed the arithmetic mean of accuracy for each of those candidate models of 1,000 iterations and identified the best two models. The performance results and comparisons are denoted in the next subsection.

4.3. Performance Comparison

Table 4 shown below denotes the mean, maximum, minimum, and standard deviation of accuracy among the five candidate models after 1,000 iterations.

Models	**Performance Attributes**			
	Mean Accuracy	Max Accuracy	Min Accuracy	Std of Accuracy
LSTM	81.06%	87.5%	70.5%	0.0284
BI-LSTM	81.67%	**91.0%**	73.0%	0.0270
GRU	81.86%	89.0%	73.0%	**0.0267**
BI-GRU	81.43%	87.0%	73.5%	0.0281
Conv-1D Vert	**82.27%**	89.0%	**75.5%**	0.0271

Table 4:performance summary of candidate models

Based on the above table, we decided to choose 2 models based on the performance of both mean accuracy and the dispersion of mean accuracy. By observing the result, GRU (Gated Recurrent Unit) and Conv-1D (Convolutional 1D) Vertical were chosen and the detailed evaluation of those models will be denoted in the next subsection.

4.4. Final Model

Depending on the performance resulting from 1,000 iterations, we noticed that the ***Conv-1D Vertical*** model and ***GRU*** model are prominent among the five candidate models. Conv-1D Vertical generates the highest mean accuracy, approximately 82.27% and GRU has the lowest standard deviation of the mean accuracy, approximately 0.0267. Besides, Bidirectional LSTM model has the highest upside with a maximum accuracy of 91% among 1,000 iterations, while Conv-1D Vertical has the highest downside with a minimum accuracy of 75.5% among 1000 iterations. Admittedly, the overall accuracy and its standard deviation are vital standards in distinguishing the performance of models. The subsequent section will analyze and disclose the strengths and weaknesses of these two models in a detailed justification.

4.4.1. Confusion Matrix and F-1 Score

For a supervised binary classification task, the ***Confusion Matrix*** is an extraordinary visualization of the performance of the deep learning algorithm. It provides a detailed breakdown of the performance of each class, which is extremely detrimental if the classes are imbalanced. A high overall accuracy may disguise the poor performance of the minority class. Thus, without the confusion matrix analysis, a binary classification model can be reasonably considered incomplete and misleading.

For comparison purposes, the graph of two confusion matrices for Conv-1D (left) and GRU (right) models are presented jointly below in Figure 10.

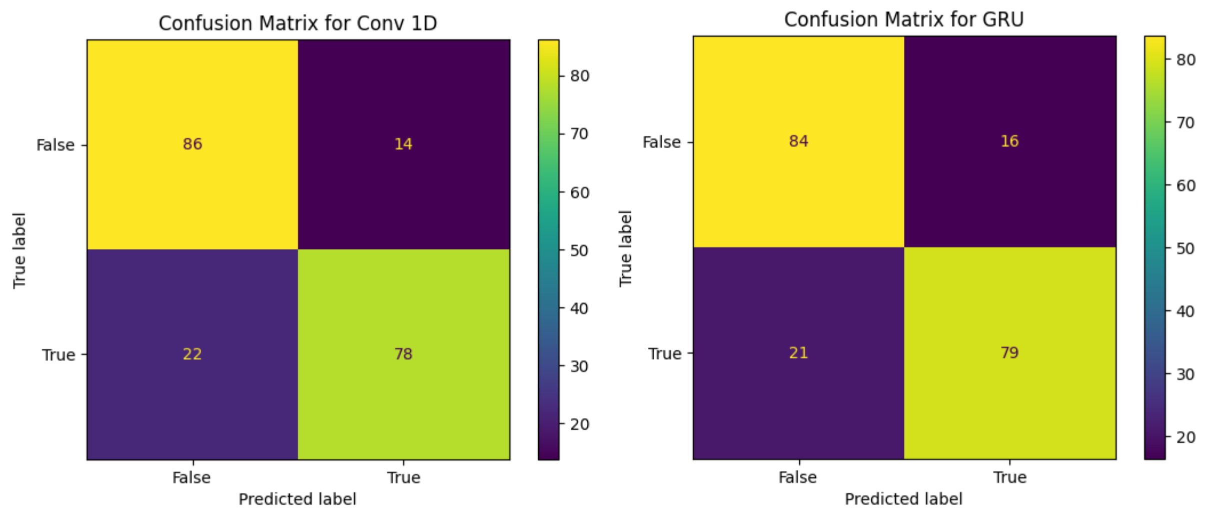


Figure 10: confusion matrices of both prediction models

The demonstration of our confusion matrix generated in Python is displaced slightly differently from the default setting. In our case, the True Positive (TP) is represented in the lower right cell, which indicates the predicted and real results are both bankrupted companies. Similarly, the False Negative (FN) is represented in the lower left cell, which indicates the predicted result is healthy and the reality is bankrupt. The False Positive (FP) is represented in the upper right cell, which indicates the predicted result is bankrupt but the reality is healthy. The True Negative (TN) is represented in the upper left cell, which indicates the predicted and the real results are both healthy companies.

In order to analyze the performance more scientifically, an F-1 score with precision and recall is a good metric to determine if the result is biased toward one class or not. The F-1 score is a measure of a model's performance that balances both precision and recall. It is the harmonic mean of precision and recall and ranges between 0 and 1, with 1 being the best possible F1 score. Since our intention was to minimize the bias in a binary-classification model, we employed the modification of the F-1 score, which is weighted for both True Positive and True Negative. It can be described by the formula below:

$$F1 = \frac{1}{2} \cdot (F1_{BC} + F1_{HC})$$

Where F-1 for each class is calculated by: BC = bankrupt company; HC = healthy company

$$F1 = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

$$precision = \frac{TP}{TP + FP} \qquad recall = \frac{TP}{TP + FN}$$

Based on the formulas presented above, the F1 scores for both of our models are presented in Table 5 (for the purpose of simplicity, we omitted the numerical calculations):

Models	**Performance Attributes**		
	F1 BC	F1 HC	Weighted F1
Conv-1D	0.8125	0.8269	**0.8197**
GRU	0.8103	0.8195	**0.8149**

Table 5:performance summary of prediction models

Following the result denoted by the table, it is reasonable to conclude that both Conv-1D and GRU display similar performance regarding weighted F-1 score. Nevertheless, Conv-1D model is slightly biased to recognizing healthy companies by observing a 1.44% higher F1 HC than F1 BC, whereas GRU is relatively balanced between the two classes, only displaying a 0.092% discrepancy between F1 HC and F1 BC.

4.4.2. ROC Curve and AUC

In addition to the confusion matrix and F-1 score valuation, ***Receiver Operating Characteristic (ROC)***, and the ***Area Under the Curve (AUC)*** are non negligible evaluation metrics for binary classification problems. ROC illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied. The ROC curve is created by plotting the true positive rate (TPR) against the false positive rate (FPR). A graph closer to the upper left corner indicates a more exceptional performance. The graph closer to the 45-degree curve in the ROC space indicates less accuracy. Note that the blue dash line shows a random classifier. The AUC ranges from 0 to 1 and a model whose prediction has 100% correctness has AUC value of 1 and 100% wrongness has AUC value of zero.

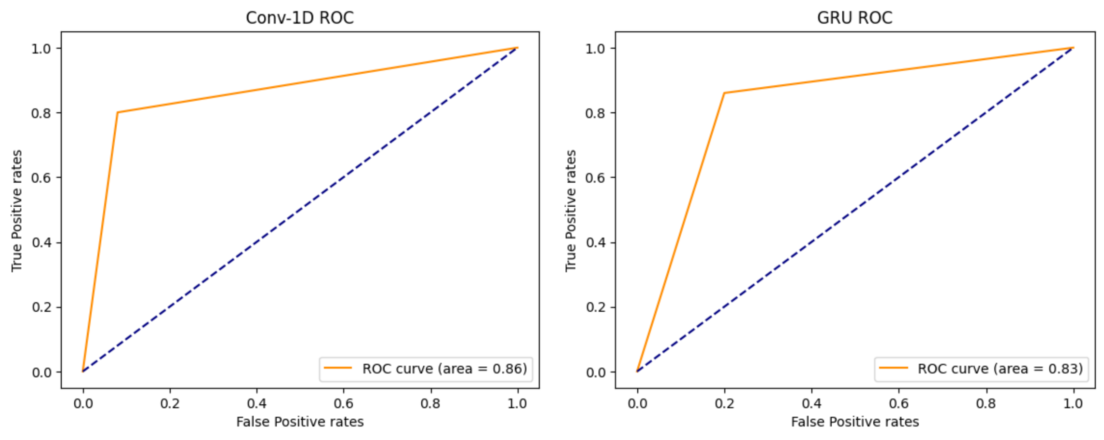


Figure 11: ROC of both prediction models

The two graphs shown above suggest that both Conv-1D and GRU models have excellency in our classification task, with a minimal higher area under curve for Conv-1D. In empirical studies, a model with an AUC higher than 0.8 is considered well-functioned. Again, based on the fact that the ROC is a trade-off between true positive rate and false positive rate for different thresholds, we observe that Conv-1D trades towards more false positive rates to true positive rates.

4.5. Model Interpretation

In principle, it can be asserted that understanding why a deep learning model performs well is a fairly challenging task. This obstacle is usually referred to as the "black box" problem of deep learning, which means that although we can infer what goes in (the input data) and what comes out (the predictions), the internal mechanisms of the model are hardly interpretable by humans due to the fact of complexity, nonlinearity, and lack of clear rules, unlike the decision tree systems. However, we tried our best to analyze the backed reasonings and deliver our judgments regarding why the models are underperforming or outperforming the others. Noteworthily, these are our interpretations, which may not represent what is actually inside the "black box".

4.5.1. The Failure of Our Conceptual Model (LSTM)

It is generally accepted that the LSTM model deals with long-term time series data better than other recurrent neural networks because of the reasons we introduced previously in the paper. Nonetheless, our dataset only consists of 5-year time series data with 28 different features, which cannot be considered long-term clearly. Thus, it is not necessary that LSTM performs better than other models. In addition, LSTM has complex models with many parameters to tune. This makes it computationally intensive to train and requires a lot of data. Notwithstanding, we just have 220 bankrupt companies to train and the paucity of sufficient data might hinder the LSTM to learn effectively. Therefore, we reckon that LSTM is overqualified in this classification task and it is a great talent gone to waste.

4.5.2. The Success of Convolution Layer (Conv-1D)

It is surprising to find out that Conv-1D has the best mean accuracy after 1,000 iterations and it is the only architecture that exceeds 82% accuracy among candidate models. Conv-1D has a relatively simpler internal structure than RNNs such as LSTM and GRU. It is commonly used also for sequential data, especially short-term sequential data, which is exactly what we desire in this project. Conv-1D layers can capture local and position-invariant features in the sequence. This can be useful when the order of the data is important, which we think is essential for binary classification. From our perspective, to conclude, the advantages of using Conv-1D layer in this project are that Conv-1D provides simpler computation and local patterns are accurately found by the convolutional layer given short-term time series.

4.5.3. The Success of Gated Recurrent Unit (GRU)

The Gated Recurrent Unit layer is quite similar to the LSTM layer, while GRU has a moderately simpler architecture than LSTM with fewer parameters. This allows it to result in faster training times and lower computational requirements, which can be particularly beneficial when dealing with short-term time series data. Furthermore, due to having fewer parameters, GRUs can be slightly less prone to overfitting compared to LSTMs. In short, our task does not require very complicated models and those models that can deal with time series with relatively simple architecture tend to outperform more complicated models.

5. Conclusion

5.1. Paper Summary

Throughout the development of the financial market, estimating and predicting the bankruptcy of a certain company is crucial in terms of investors' confidence in the whole economy, policy-making and decisions, credit and derivative market analysis, etc. Consequently, this paper strives to utilize RNN's ability to analyze time series to predict the bankruptcy of companies given market, fundamental, and stock data. After performing necessary data preprocessing including transpose and normalization, a conceptual model of LSTM with hyper-parameters of tanh and PReLU activation functions, Adam optimizer, BinaryCrossEntropy loss function, dropout regularization, callback function, and 1024 batch size, was built based on our logic and previous empirical studies for the purpose of comprehension. Subsequently, two prediction models were selected from a pool of 5 conceptual models based on the mean accuracy and a detailed performance analysis regarding accuracy, precision, ROC, and F1 score was conducted. Besides the summarized procedures of our model, it is also worth noting that the final two prediction model choices deviated from our previous conceptual models, primarily due to the fact that our dataset only contains 5-year time step for each block of data, which does not suit for the strength of LSTM; instead, convolutional 1D vertical and GRU models outperformed and we assume such outperformance was due to their ability to deal with shorter time series data. Moreover, the performance, mean accuracy in particular, of all the models we built cannot be further improved even though a fair amount of effort has been made, including but not limited to tuning the architecture or hyper-parameter, to build our models. Consequently, we assume that the 28 attributes provided in the original dataset were not enough to fully capture the effective features of bankruptcy and thus in order to achieve a higher mean accuracy, other attributes that would potentially and effectively affect bankruptcy should be found.

5.2. Other Notable Endeavors

As it is stated in the previous subsection, a fair amount of effort and time have been devoted by us to finally achieving these two prediction models. Some intuitive and reasonable endeavors are presented below, which we believe will make our model able to deal with real-world scenarios:

1) To keep the realistic features of our model, instead of employing fixed train, test, and validation data to evaluate the performance of our models, we evaluate the model by iterating 1000 times, within which the choice of train, test, and validation data are completely random. The primary rationale of this method is based on the fact that in reality, the income training and testing datasets are random and unknown. Thus, instead of testing the performance of our model on a single and fixed dataset, it is reasonable to test the performance of the models on their true ability to handle this kind of problem in the real world.
2) After over 10,000 trial-and-error in all conceptual and prediction models, we concluded that since the dataset provided only consists of a 5-year timestep, a relatively simple architecture (in terms of the number of layers and neurons) should be employed to achieve a higher mean accuracy. During the process, we tried to increase the complexity of the model and employed a higher regularization rate and constraints, hoping the complexity will be monitored and regularized by a tighter regularization. However, we eventually found out that models with one RNN layer with neurons less than 128 are more than enough to reach a higher mean accuracy than more complex models.
3) During the model-building process, we devoted a great amount of time to researching and choosing activation functions in Dense layers. In our previous studies, instead of trying various advanced activation functions, "relu" was employed as default most of the time on Dense layers. However, in order to make our models more explainable and scientific, instead of acquiescing "relu" as a black box and default function, we researched through several research papers on advanced activation functions and performed empirical experiments on most of the functions, and eventually arrived to the destination that "PReLU" is the best choice in our model.

5.3. Limitations and Future Improvements

Despite the fact that our two prediction models achieve magnificent performances in predicting whether companies are going to default or stay healthy in terms of prediction accuracy, some limitations and weaknesses still exist in the models and future improvements could be implemented. We acknowledged and enumerated the following drawbacks as well as potential future improvements regarding each drawback:

A. In accordance with the confusion matrix and weighted F-1 score described in section 4.4.1, the Conv-1D model still has a slightly biased tendency in correctly predicting healthy companies due to the fact that the F-1 score for healthy companies is approximately 1.44% higher than it for bankrupt companies. This potential risk could be reduced by manually changing the class weights for the loss functions. In other words, our training model uses an auto-weight function that automatically gives equal weight to the two classes and the model may take advantage of increasing the bankrupt companies, for instance using the weights {0: 0.50, 1: 120} instead of using the auto weight of {0: 0.50, 1: 84.20} shown in section 2.2.4. Such a change, we believe, would emphasize more on the bankrupt companies during the training phase so that the performance of the model tilted to the bankrupt class a little bit.
B. We did not calculate the SHAPLEY values, which have the ability to interpret the contribution of each attribute to the prediction for a single instance. We are clueless as to which of the 28 attributes contributes the final prediction the most or which are not contributing at all. The SHAP value is an important tool for promoting transparency and trust in the deep learning model. If we artificially eliminate some attributes that are not contributing to the prediction based on SHAP values, the model performance might make huge progress.
C. In this paper, we only focused on the networks (RNN and CNN-1D) that theoretically deal with time series better than other architectures. We didn't pay attention to other advanced algorithms or layers such as Random Forest, XGBoost, or attention layers used in transformers. Such a limitation restrains the possibility of more reliable performance for this project. We cannot assure that those advanced techniques would generate much higher accuracies on another level, but such an effort is worth experimenting in further research.
D. Finally, the most conspicuous deficiency in the paper is that all the provided data is outdated. The companies' financial data is way different than many years ago. The prediction results, thereby, generated from this deep learning model are not exceptionally representative of modern companies in the 2020s. Nevertheless, a similar methodology could be applied using more recent financial data and the resulting model would have exceedingly higher predictive power.

6. References

[1] Kingma, Diederik & Ba, Jimmy. (2014). *Adam: A Method for Stochastic Optimization.* International Conference on Learning Representations.

[2] Ruby, Usha & Yendapalli, Vamsidhar. (2020). *Binary cross entropy with deep learning technique for Image classification.* International Journal of Advanced Trends in Computer Science and Engineering. 9. 10.30534/ijatcse/2020/175942020.

[3] De, Richard & Lyle, Veaux & Ungar, H.. (2000). *Multicollinearity: A tale of two nonparametric regressions*. 10.1007/978-1-4612-2660-4_40.

[4] Bai, Yuhan. (2022). *RELU-Function and Derived Function Review*. SHS Web of Conferences. 144. 02006. 10.1051/shsconf/202214402006.

[5] He, Kaiming & Zhang, Xiangyu & Ren, Shaoqing & Sun, Jian. (2015). *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. IEEE International Conference on Computer Vision (ICCV 2015). 1502. 10.1109/ICCV.2015.123.

[6] Clevert, Djork-Arné & Unterthiner, Thomas & Hochreiter, Sepp. (2015). *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)*. Under Review of ICLR2016 (1997).

[7] Bengio, Yoshua. (2012). *Practical recommendations for gradient-based training of deep architectures*. Arxiv.

[8] Chollet, Francois. *Deep Learning with Python*. Manning, 2018.

[9] Xu, Bing & Wang, Naiyan & Chen, Tianqi & Li, Mu. (2015). *Empirical Evaluation of Rectified Activations in Convolutional Network.*

[10] Kukacka, Jan & Golkov, Vladimir & Cremers, Daniel. (2017). *Regularization for Deep Learning: A Taxonomy.*

[11] Juan Pablo Bonfrisco Ayala, Javier de Vicente Moreno. (2019). *APPLYING DEEP LEARNING TECHNIQUES TO PREDICT THE BANKRUPTCY OF AN ENTERPRISE*

[12] Phaisangittisagul, Ekachai. (2016). *An Analysis of the Regularization Between L2 and Dropout in Single Hidden Layer Neural Network*. 174-179. 10.1109/ISMS.2016.14.

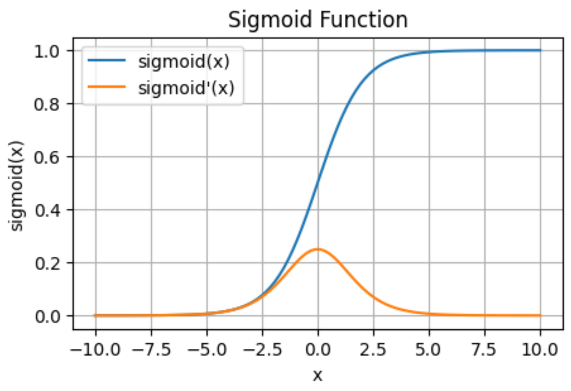
[13] Passricha, Vishal & Aggarwal, Rajesh. (2019). *A Hybrid of Deep CNN and Bidirectional LSTM for Automatic Speech Recognition*. Journal of Intelligent Systems. 29. 10.1515/jisys-2018-0372.

[14] Dey, Rahul & Salem, Fathi. (2017). *Gate-Variants of Gated Recurrent Unit (GRU) Neural Networks.*

[15] Gupta, Akhilesh & Tatbul, Nesime & Marcus, Ryan & Zhou, Shengtian & Lee, Insup & Gottschlich, Justin. (2020). *Class-Weighted Evaluation Metrics for Imbalanced Data Classification.*

7. Appendix

Figure 1 & 2: graph of sigmoid (left) and tanh (right) functions and their derivatives



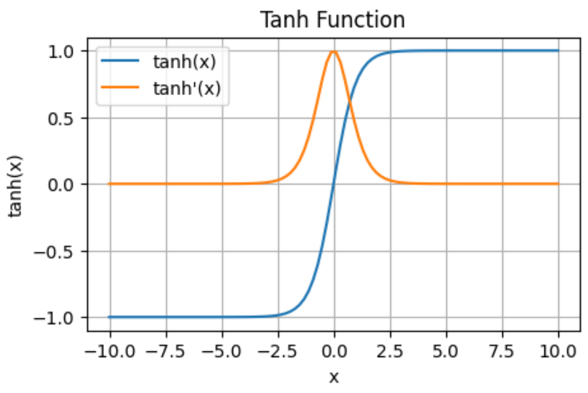
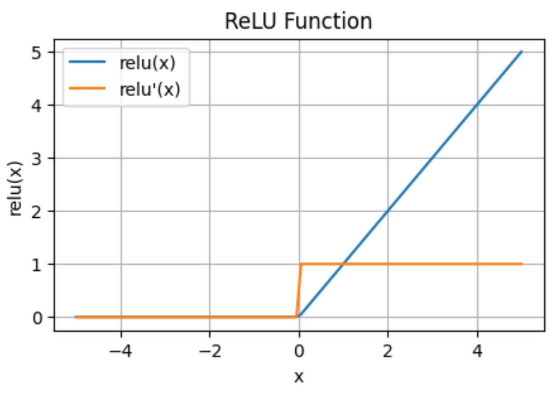
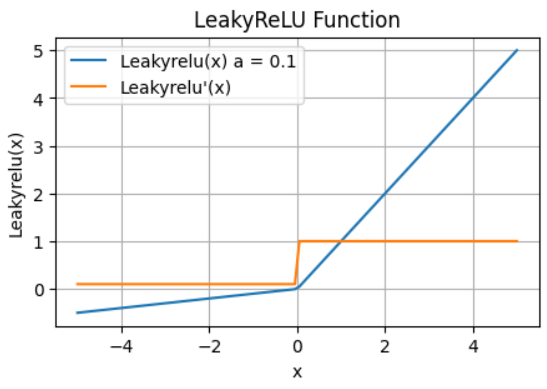
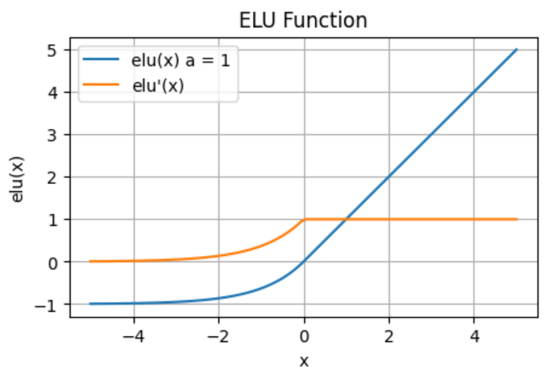


Figure 3-6: graph of relu, leakyrelu, elu, and prelu functions and their derivatives







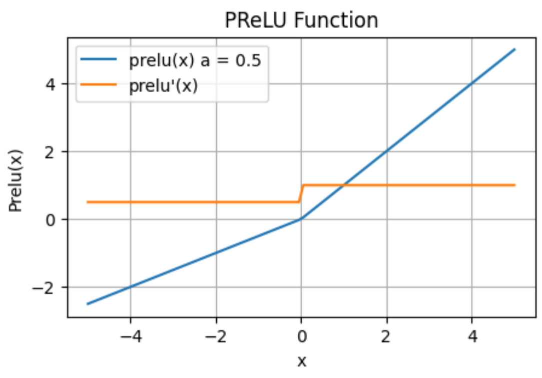


Figure 7: graph of overfitting

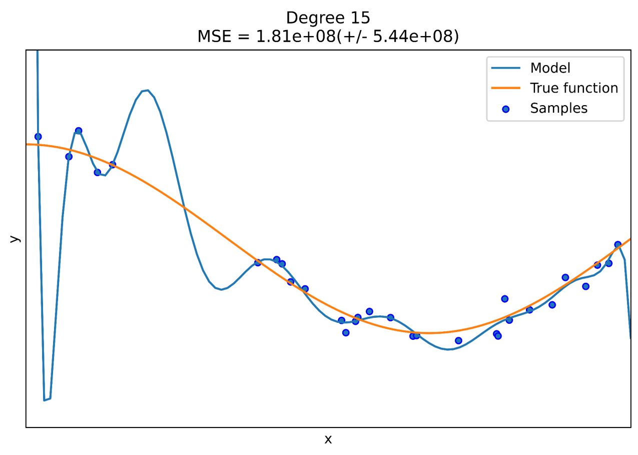


Figure 8: graph representation of model architecture

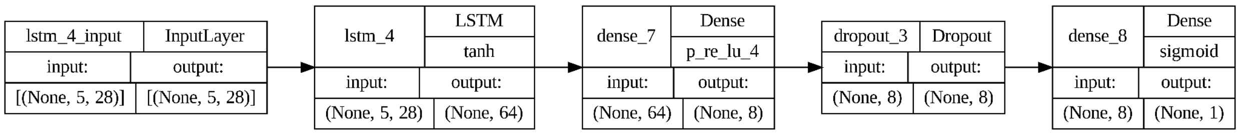
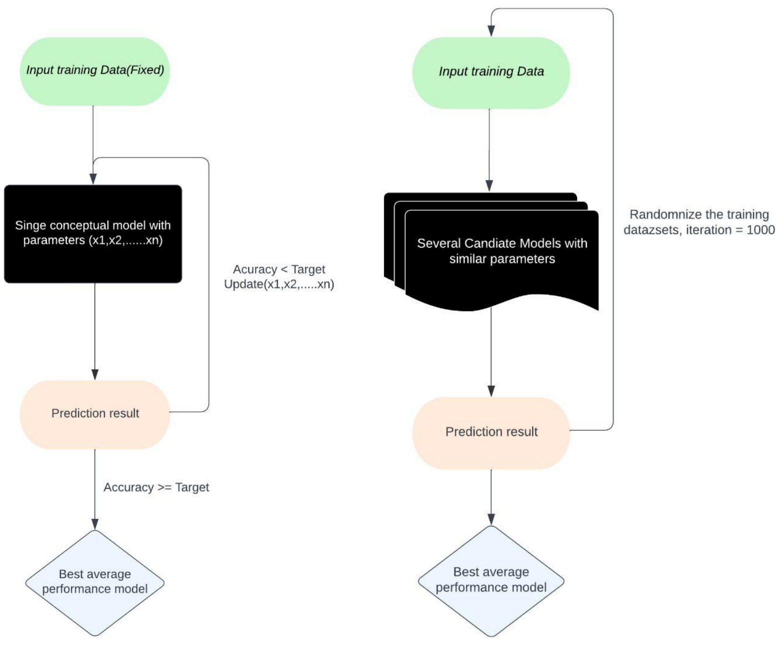


Figure 9: flow map of prediction models selection process


Input training Data(Fixed)
Singe conceptual model with parameters (x1,x2,......xn)
Acuracy < Target
Update(x1,x2,.....xn)
Prediction result
Accuracy >= Target
Best average performance model
Input training Data
Several Candiate Models with similar parameters
Randomnize the training datazsets, iteration = 1000
Prediction result
Best average performance model

Code 1: code of building prediction models

<u>LSTM</u>

```
model = keras.Sequential()
model.add(keras.layers.LSTM(64,input_shape=(5,train_x.shape[2]),activati
on = 'tanh', dropout = 0.125))
model.add(keras.layers.Dense(32, activation = 'PReLU'))
model.add(keras.layers.Dropout(0.125))
model.add(keras.layers.Dense(1,activation = 'sigmoid'))
model.compile(loss= tf.keras.losses.BinaryCrossentropy(), optimizer =
tf.keras.optimizers.Adam(), metrics = ['accuracy'])
try1 = model.fit(train_x, train_y, validation_data=(val_x, val_y),epochs
= 40, batch_size = 1024, callbacks =
[es_callback],class_weight=class_weight_auto,verbose = 0)
test_loss, test_acc = model.evaluate(test_x,test_y,batch_size=256)
```

BI-LSTM

```
model1 = keras.Sequential()

model1.add(keras.layers.Bidirectional(keras.layers.LSTM(50,input_shape=(
5,train_x.shape[2]),activation = 'tanh', dropout =
0.1,kernel_regularizer = keras.regularizers.L1(0.00007))))
model1.add(keras.layers.Dense(25, activation = 'PReLU'))
model1.add(keras.layers.Dropout(0.2))
model1.add(keras.layers.Dense(5, activation = 'PReLU'))
model1.add(keras.layers.Dropout(0.2))
model1.add(keras.layers.Dense(1,activation = 'sigmoid'))
model1.compile(loss= tf.keras.losses.BinaryCrossentropy(), optimizer =
tf.keras.optimizers.Adam(), metrics = METRICS)
try2 = model1.fit(train_x, train_y, validation_data=(val_x,
val_y),epochs = 40, batch_size = 1024, callbacks =
[es_callback],class_weight=class_weight_auto,verbose = 0)
```

GRU

```
model = keras.Sequential()
model.add(keras.layers.GRU(64,input_shape=(5,train_x.shape[2]),activatio
n = 'tanh', dropout = 0.125))
model.add(keras.layers.Dense(32, activation = 'PReLU'))
model.add(keras.layers.Dropout(0.125))
model.add(keras.layers.Dense(1,activation = 'sigmoid'))
model.compile(loss= tf.keras.losses.BinaryCrossentropy(), optimizer =
tf.keras.optimizers.Adam(), metrics = ['accuracy'])
try1 = model.fit(train_x, train_y, validation_data=(val_x, val_y),epochs
= 40, batch_size = 1024, callbacks =
[es_callback],class_weight=class_weight_auto,verbose = 0)
test_loss, test_acc = model.evaluate(test_x,test_y,batch_size=256)
```

BI-GRU

```
model = keras.Sequential()
```

```
model.add(keras.layers.Bidirectional(keras.layers.GRU(64,input_shape=(5,
train_x.shape[2]),activation = 'tanh', dropout = 0.125)))
model.add(keras.layers.Dense(32, activation = 'PReLU'))
model.add(keras.layers.Dropout(0.125))
model.add(keras.layers.Dense(1,activation = 'sigmoid'))
model.compile(loss= tf.keras.losses.BinaryCrossentropy(), optimizer =
tf.keras.optimizers.Adam(), metrics = ['accuracy'])
try1 = model.fit(train_x, train_y, validation_data=(val_x, val_y),epochs
= 40, batch_size = 1024, callbacks =
[es_callback],class_weight=class_weight_auto,verbose = 0)
test_loss, test_acc = model.evaluate(test_x,test_y,batch_size=256)
```

Conv-1D vertical

```
model = keras.Sequential()
model.add(keras.layers.Conv1D(filters = 128,kernel_size =
5,input_shape=(28,5),activation = 'tanh',padding =
'same',kernel_regularizer = keras.regularizers.L1(0.00007)))
model.add(keras.layers.Dropout(0.125))
model.add(keras.layers.MaxPooling1D(pool_size = 2))
model.add(keras.layers.Conv1D(filters = 64,kernel_size = 4,activation =
'tanh',padding = 'same',kernel_regularizer =
keras.regularizers.L2(0.00007)))
model.add(keras.layers.Dropout(0.125))
model.add(keras.layers.MaxPooling1D(pool_size = 2))
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(32, activation = 'PReLU'))
model.add(keras.layers.Dropout(0.125))
model.add(keras.layers.Dense(4, activation = 'PReLU'))
model.add(keras.layers.Dropout(0.125))
model.add(keras.layers.Dense(1,activation = 'sigmoid'))
model.compile(loss= tf.keras.losses.BinaryCrossentropy(), optimizer =
tf.keras.optimizers.Adam(), metrics = METRICS)
try1 = model.fit(train_x.transpose(0,2,1), train_y,
validation_data=(val_x.transpose(0,2,1), val_y),epochs = 40, batch_size
= 1024, callbacks = [es_callback],class_weight=class_weight_auto,verbose
= 0)
```