Deep Learning Technique for Credit Card Default Prediction

Author: Kaylum Smith

Abstract:

This report investigates the application of deep learning techniques for predicting credit card defaults (CCD). Correctly predicting credit card default is crucial for banks. I learned and tested multiple approaches to solve this problem, primarily using a deep neural network (DNN), and documented the results as I went along. The study processed credit card default data, including relevant data about each customer, to train the models. The final model yielded very strong results in its ability to learn from the data set.

Introduction:

Summarise and highlight

The importance of accurately predicting credit card defaults (CCD) is covered heavily in machine learning and deep learning. Traditional statistical methods are being replaced with optimal solutions from different learning approaches, such as Deep Neural Networks (DNNs), Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and feedforward Neural Networks (FNNs).

This project aimed to detail the different applications of deep learning algorithms to the CDD data set, comprised of 3000 individuals and their relevant personal details. This included LIMIT_BAL, sex, education, marriage, age, pay, bill amount and whether they had a credit card default.

The objective was to develop an algorithm with the best predictive model to determine next month's payment likelihood accurately. These predictions are very valuable for banks and businesses when evaluating a person's creditworthiness or eligibility for a loan. The final algorithm was optimised using the DNN model, which achieved an Accuracy of 82% and a loss of 44%, showing promising results for correctly predicting if a person successfully paid off the subsequent credit payment.

Finally, describe how the report is organised.

The report is broken down into an introduction, Proposed Method, Experimental Results, and summary.

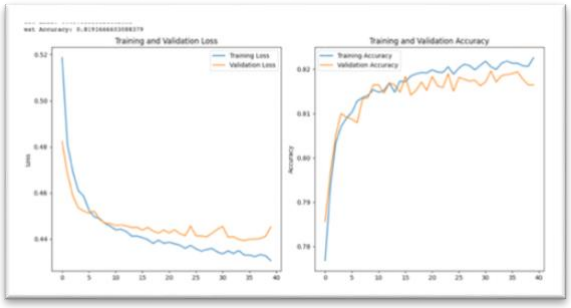
Proposed Method:

Proposed deep learning methodology:

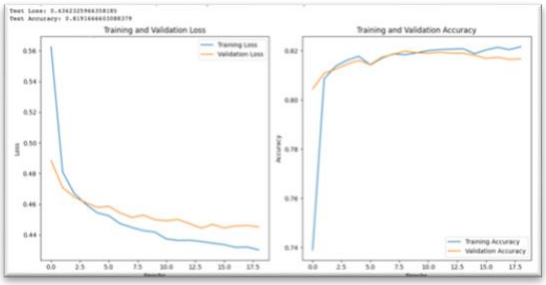
Given that the dataset is for binary classification tasks where the prediction is either 0 or 1, a Deep Neural Network (DDN) was used as it is better at handling non-temporal data, unlike CNNs, which excel in spatial data recognition and RNNs, which are tailored for sequential data processing.

The DNN I implemented has multiple dense, fully connected layers with different numbers of neurons; the first layer has 64 neurons, the next hidden layer has 32 neurons, and the output layer has one.
Each hidden layer uses the Relu activation function over others due to its lower loss and higher accuracy. Still, the main reason was the resulting "training and validation loss" and "training and validation accuracy" graphs, which ended up smoother with fewer fluctuations. Finally, the output layer uses the sigmoid activation function.

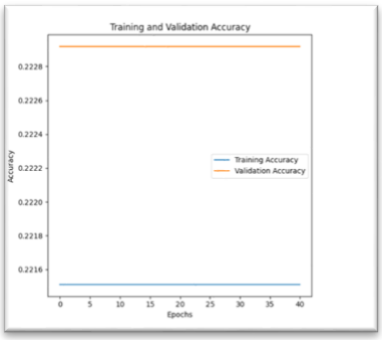
Using: sigmoid, relu, sigmoid



using: relu relu, sigmoid



E.g. using softmax output layer




```python
# Define the deep neural network model
model = Sequential([
    Dense(64, activation='relu', input_shape=(X_train.shape[1],), kernel_regularizer=l2(0.001)),
    Dropout(0.3),
    Dense(32, activation='relu', kernel_regularizer=l2(0.001)),
    Dropout(0.3),
    Dense(1, activation='sigmoid', kernel_regularizer=l2(0.001))
])

# Compile the model
model.compile(optimizer=optimizers.Adam(), loss='binary_crossentropy', metrics=['accuracy'])
```

Image: final model definition

The model is trained using a fit() method, one of the most widely used methods, especially when using Keras.

Data pre-processing steps:

First, the data was loaded from the credit card default dataset from Excel using the `pd.read.excel()`. The data is further separated into input features (X) and the target variables (Y). This is done so the deep learning algorithm can start the learning process.

Then, using the' StandardScaler () ' function, a standardizer was applied to the input features to ensure each feature had a mean of 0 and a standard deviation of 1, enhancing the model's convergence efficiency.

Finally, Training is split into the training and validation sets using the `train_test_split()` function, with 0.8 training and 0.2 validation. Increasing or decreasing the training and validation causes a higher loss score and worse accuracy, and the confusion matrix has a higher incorrect prediction ratio.

Experimental Results

Description hyperparameter settings

An early stooping was implemented, pre-emptively ceasing further epochs when no improvement was observed after five consecutive iterations. The epochs are set to 50 as I have an early stopping set, and the algorithm rarely uses more than 45 epochs due to early stopping. Additionally, a reduced number of epochs contributed to a better balance between training and validation loss, avoiding fluctuations that indicated potential overfitting.

```
# Train the model
history = model.fit_generator(X_train, Y_train, epochs=50, batch_size=256, validation_split=0.2, verbose=1, callbacks=[early_stopping])
```

Image: final model training

Image with early stopping few epochs.

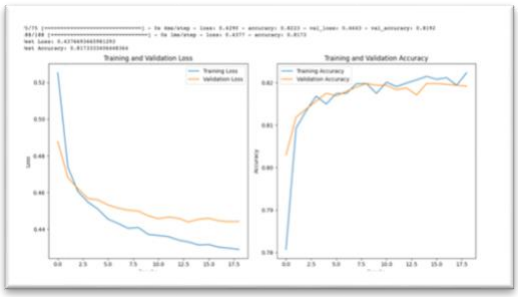
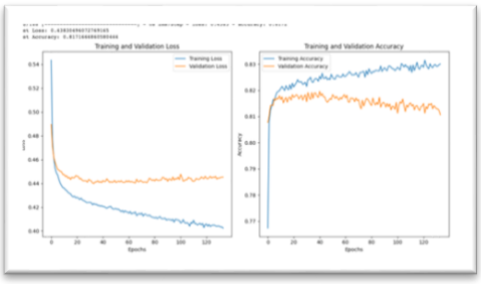


Image with higher epochs and later stopping



The batch size 256 was implemented, as smaller sizes resulted in reduced validation accuracy and greater loss.

Image normal 256 batch size

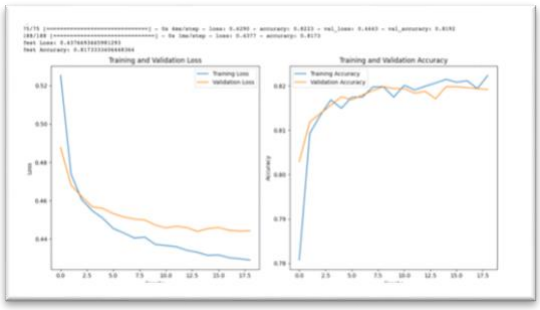
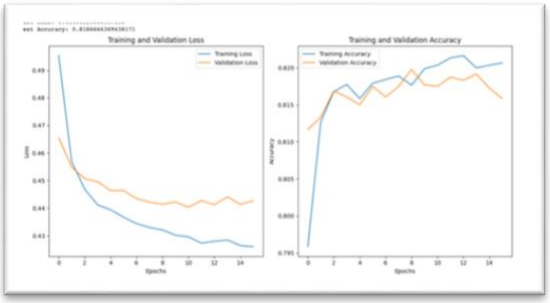
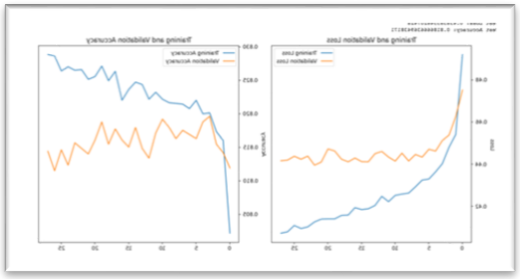


image much lower batch size

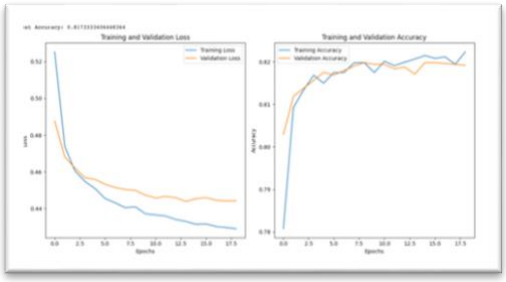


A validation split of 0.2 provided the most reliable performance, with more extensive testing sets leading to poorer outcomes.
Then, Dropout layers were inserted to help with overfitting, and the results show that the most effective configuration for the dense layers was 0.3. This percentage was chosen after implementing the L2 regularizer, which dramatically improved the training and validation loss graph but increased overfitting in training and validation accuracy. This approach outperformed other dropout rates, which either led to higher volatility or overfitting, as evidenced by the accompanying graphs.

Higher dropout 240, 120, 1.



Best dropout of 64, 32, 1



The model is compiled using the Adam optimizer instead of SGD or any other due to its adaptive learning rate over others, which often leads to faster convergence. The binary cross-entropy loss function was selected as the most appropriate function for the binary classification task, and accuracy was chosen as the performance metric to track it deemed the standard. Using SGD causes more overfitting.

```
# Compile the model
model.compile(optimizer=optimizers.Adam(), loss='binary_crossentropy', metrics=['accuracy'])
```

Image using Adam

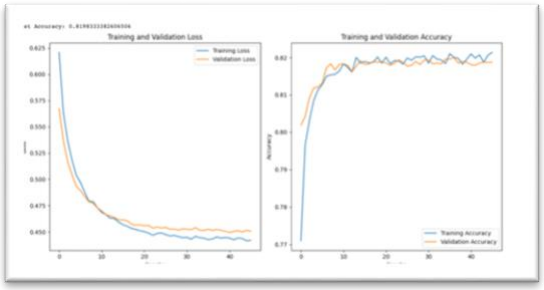
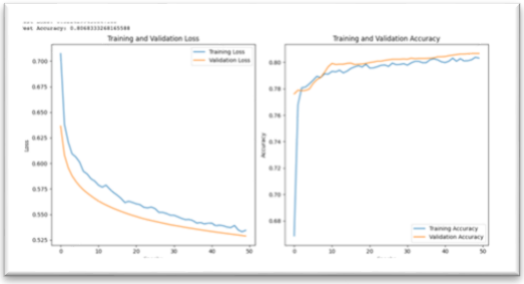


image using SGD



A random state set to 42 was the best as it increased the accuracy, while others decreased the graph precision and created more volatility, such as 80, which caused some effects, as shown below.

Description evaluation process

The test loss score is the value of the loss on the test dataset. It indicates how well the model's predictions match the actual labels in the test set. Lower values indicate better performance. The accuracy score represents the correctly classified instances in the test dataset. It means the overall correctness of the model's predictions on unseen data. Higher values indicate better performance. In Addition, the confusion matrix and classification report help indicate essential information, such as a comprehensive understanding of the model's performance, particularly in terms of precision, recall, and F1-score.

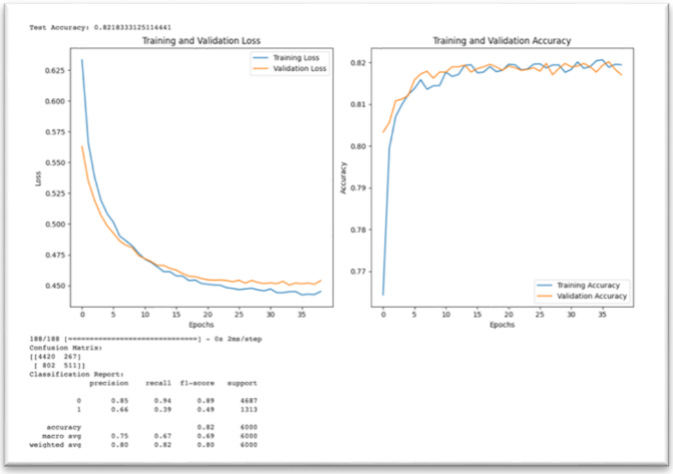
Evaluation metrics focused on accuracy, precision, recall, and the F1-score. The model's accuracy on the test set reached 0.82, indicating a high level of predictive reliability. Precision was recorded at 0.67, suggesting a substantial rate of true positives among predicted defaults. However, recall was observed at 0.32, reflecting a need for improvement in identifying actual defaults.

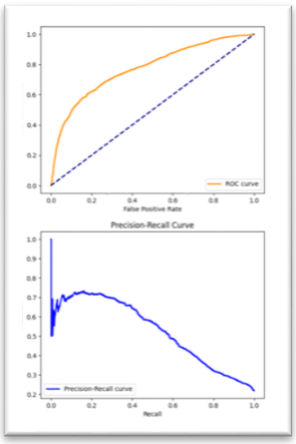
A straightforward evaluate() method was implemented to evaluate the trained model, and multiple metrics, such as test loss and accuracy, confusion matrix, and classification report, were implemented to better understand my algorithm.

```
 Evaluate the model on the test set
est_loss, test_accuracy = model.evaluate(X_test, Y_test, verbose=1)
rint("Test Loss:", test_loss)
rint("Test Accuracy:", test_accuracy)
```

Image: test loss, test accuracy

Visualization played a pivotal role in interpreting the model's learning dynamics. I plotted training, validation loss, and accuracy over epochs to monitor for signs of overfitting. The introduction of ROC and Precision-Recall curves provided additional clarity on the model's prediction capabilities and classification thresholds.
The model analysis shows the “training and validation loss” and “training and validation accuracy” graphs; the trends over the epochs give us insight into overfitting or undefining problems.



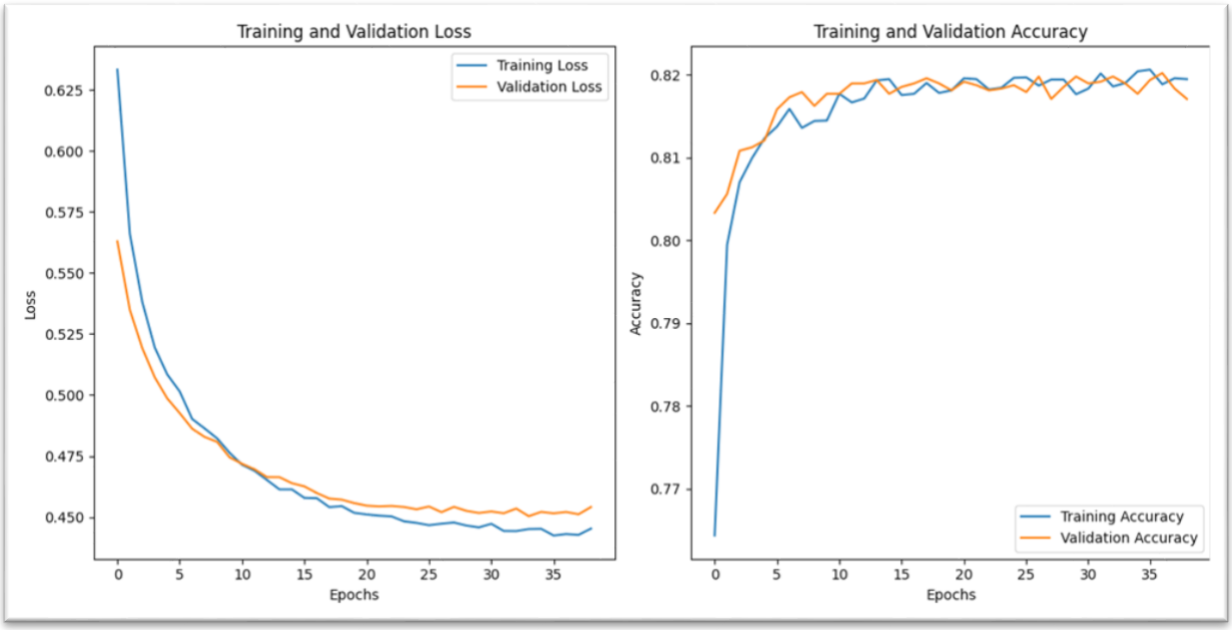


.

obtained results

The final DNN algorithm produced a test accuracy of 82% and a loss of 44%; this shows its promise in accurately predicting the dataset over what the test was repeated to show its effectiveness.

In the training and validation loss graph, both the training loss and the validation loss decrease sharply at the beginning and then plateau, converging to similar values. This convergence suggests that the model is learning effectively without overfitting. Then, the Training and Validation Accuracy Graph shows that the training accuracy rises sharply initially before plateauing, much like the loss graph. The validation accuracy also increases and is close to the training accuracy after the first epoch, telling us the model is suitable for generalisation from the training data to unseen validation data.
Though there is a slight gap between the two, the validation accuracy closely follows the training accuracy, which is a positive indicator of model performance.
Both graphs display characteristics of a well-fitting model. The loss is minimised without a growing gap between training and validation loss. The accuracy stabilises at a high level with close alignment between training and validation metrics.



Now, the classification reports show the accuracy, with a macro average precision of 0.75 and recall of 0.67, showing good performances. The weighted average gives more weight to the majority and shows a precision of 0.80 and recall of 0.82, suggesting the model performs well on the majority class but less on the minority class.

The model has high overall accuracy but shows a discrepancy between the performance of the two classes, with class **1** (defaults) having lower recall. The high number of false negatives (802) for class **1** suggests that the model tends to predict non-defaults more than defaults, which may be a concern in a real-world setting where predicting actual defaults is crucial.

This is further shown in the confusion matrix, where the matrix indicates the model is adept at identifying non-defaults with 4420 True Negatives.

It also shows many False Negatives with 802 instances where defaults were misclassified as non-defaults. Conversely, True Positives were lower at 511. The model's tendency to overlook defaults, suggested by the high false negatives, may indicate a class imbalance, indicating insufficient learning from the minority class.

```
188/188 [==============================] - 0s 2ms/step
Confusion Matrix:
[[4420  267]
 [ 802  511]]
```

Image: final confusion matrix

```
Classification Report:
              precision    recall  f1-score   support

           0       0.85      0.94      0.89      4687
           1       0.66      0.39      0.49      1313

    accuracy                           0.82      6000
   macro avg       0.75      0.67      0.69      6000
weighted avg       0.80      0.82      0.80      6000
```

Image: final classification report

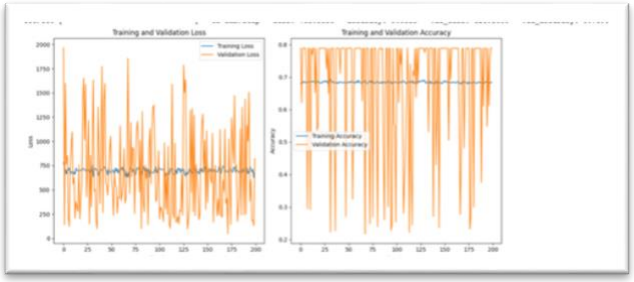
Some of the challenges I found along the way when implementing the DNN were;
Parameter Optimization: Initial models had poor performance, prompting a series of experiments with different network architectures, learning rates, and regularization techniques. Through testing, I identified a configuration that minimised loss without overfitting, as evidenced by stable validation loss trends.

Model Generalizability: It was essential to ensure that the model generalised well to unseen data. Techniques such as dropout and early stopping were utilised to enhance the model's generalisation ability, which was supported by the convergence of training and validation loss curves.

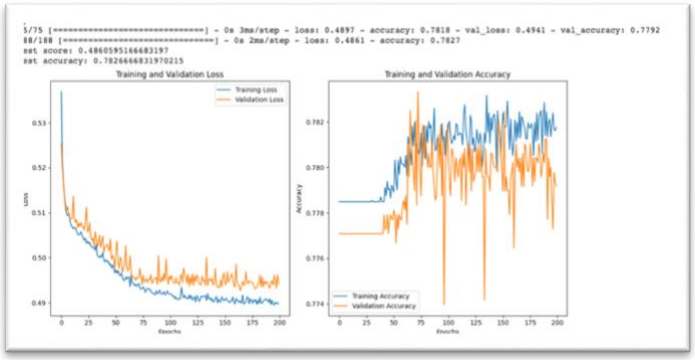
Some of the challenges I have yet to find an excellent solution to;
Class Imbalance: The dataset shows a significant imbalance between the default and non-default classes. To address this potential issue, SMOTE (Synthetic Minority Over-sampling Technique) could be implemented to balance the dataset, resulting in a fairer representation of classes. However, I could not achieve the desired result when implementing this.

Progress over time

To achieve these results, many iterations and practices of using different learning algorithms had to be implemented; the first iteration was based on the week one workshop for processing the mist dataset. The results ended up with extreme fluctuations in both training and validation loss graph.



Then, after a few more iterations, it still showed some obvious problems, especially in the training and validation accuracy graph model's unstable ability to generalize. This could also be a sign of overfitting.



These examples show the slow process of integrating new solutions in each iteration until I get the final model as shown earlier.

Summary

In summary, the study demonstrates the effectiveness of the proposed deep learning technique for predicting credit card defaults, achieving promising results in accuracy, precision, and recall metrics. However, I encountered limitations such as class imbalance and parameter tuning challenges, suggesting the need for further research in these areas.

References

- Ertel, Wolfgang, author. Introduction to Artificial Intelligence, 2017
- Doug Steen, Precision-Recall Curves. https://medium.com/@douglaspsteen/precision-recall-curves-d32e5b290248
- Jason Brownlee, ROC Curves and Precision-Recall Curves for Imbalanced Classification. https://machinelearningmastery.com/roc-curves-and-precision-recall-curves-for-imbalanced-classification/
- Mostafa Ibrahim, A Deep Dive Into Learning Curves in Machine Learning. https://wandb.ai/mostafaibrahim17/ml-articles/reports/A-Deep-Dive-Into-Learning-Curves-in-Machine-Learning--Vmlldzo0NjA1ODY0
- Jason Brownlee, Your First Deep Learning Project in Python with Keras Step-by-Step. https://machinelearningmastery.com/tutorial-first-neural-network-python-keras/