DEEP LEARNING – DENSE NEURAL NETWORK (DNN) FOR LOAN INDUSTRY

Mar 2024

Written by: Thao Phuong Vu, Rania Irfan, Kota Kota, Yanmei Li, Dhanush Krishna Narayanasamy, Valerio Stracqualursi

Table of Contents

1. Executive summary

The study investigates the effectiveness of Deep Learning (DL) in predicting loan defaults for a consumer lending company to improve operations.

Rigorous data preparation techniques were employed to ensure model effectiveness based on a 50,000 observations dataset. We addressed missing values by removing columns with data gaps, created a new column to better categorize regions, applied label encoding to convert ordinal categorical variables and one-hot encoding for nominal categorical variables, and converted date columns into a numerical format.

By implementing and optimizing a 2-layer Dense Neural Network (DNN) model we obtain 96.8% accuracy in predicting bad loans on the test set with 94.6% precision, 84.1% recall and 89% F1-score. Based on these results, limitations, recommendations and further research questions have been discussed.

2. Introduction

As a consumer lending company, we are facing challenges such as loan approval efficiency, risk assessments, and customer satisfaction. To address them, our department will use analytics to improve company operations and overall performance.

Besides, Abakarim (2018) addressed that loan approval is a binary classification problem, a set of loan applicant's data is analyzed and classified in “good” or “bad” risk. When it comes to lending risks, there are cases in which consumer is not able to pay loan back, these are loan default cases (Kumar et al.,2018).

The identification of such applicants using DL is the aim of our research. We built DNN model on loan dataset to predict bad loans to build an automated solution to loan applications. DNN consists of densely connected layers where each neuron adjusts through learning, receiving inputs from and providing inputs to all neurons in adjacent layers (Nazari and Yan, n.d). We performed data preprocessing on historical data to train and evaluate our model.

3. Motivation

Nowadays, there are many risks associated with lending money, and one of the duties of a consumer lending company, as a financial institution, is to reduce lending risks (Ince, H. and Aktan, B., 2009).

Recently, banks have employed credit counting models to categorize good loans and bad loans. However, as banking processes evolved, Lee et al. (2006) discovered many drawbacks in the credit scoring model such as insufficient economic and manpower.

Recently, DL methods have been used in financial applications (Kvamme et al., 2018). According to Byanjankar (2015), several approaches have been used for predictions, but ML and NNs methods indicate a more successful trend. Therefore, we will use DNN for loan prediction. Instead of traditional credit scoring model, DNN would help lending companies reduce manual processing

time, costs, and improve overall efficiency by providing precise and reliable risk assessments to help banks and lending companies manage risks and reduce loan default losses.

4. Data

The provided loan dataset of a consumer lending company consists of 50,000 observations explained by 53 features detailing borrowers' credit scores, loan status, and recovery efforts for missed payments from 2012-2013.

Data preparation

Basic feature eengineering was performed on data to make it suitable for DNN model. Noise variables that provided little predictive value for our target variable ‘loan_is_bad’, such as 'id', 'emp_title', 'desc', and 'policy_code', were removed.

- *Missing Values*

The missing values were addressed by removing columns with data gaps e.g., ‘next_pymnt_d’, ‘tot_coll_amt’, ‘total_credit_rv’ etc., particularly those with no direct influence on the loan outcome.

The ’mths_since_last_delinq’, ‘mths_since_last_major_derog' and ‘mths_since_last_record’ columns had significant missing entries. Missing values in these columns were replaced with binary indicators: '0' for no delinquency or derogatory marks, and '1' for any past incidents, maintaining information.

- *‘region’ Column*

A new column ‘region’ was created to categorize ‘addr_state’ according to the geographical regions - Northeast, Midwest, South, and West.

- *Label Encoding*

Label encoding was applied to ordinal categorical variables such as 'grade' and ‘sub_grade, converting them into numeric values and preserving their ordinal nature.

- *One-Hot Encoding*

For nominal categorical variables such as ‘purpose’, ‘verification_status’, ‘home_ownership’, ‘loan_status’, ‘region’, one-hot encoding was employed.

- *Date Conversion*

Dates were converted into continuous variables by calculating days from a reference date (01-Jan-2024).

- *Scaling*

All columns were standardized using StandardScaler.

5. Model Architecture

Data is split (stratified) into training (70%), test (20%) and validation (10%) sets. The neural network is implemented using TensorFlow's Keras API.

Parameter Configuration:

- The model is compiled with AdamW optimizer with weight decay 0.005 to prevent overfitting, and a learning rate of 0.001.
- Binary cross-entropy is used as loss function, suitable for classification tasks (Boer et al. 2005).
- ReLU activation for hidden layers helps in overcoming the vanishing gradient problem (Nair, 2010).
- The output layer consists of a single neuron with a sigmoid activation function, suitable for binary outcomes.
- A depth-2 neural network with a suitable activation function can help predict an outcome to any desired accuracy (Lu et al., 2010).
- The number of neurons (width of the network) needed can grow exponentially with the complexity of the function or the dimensionality of the input space (Lu et al., 2010).
- A dropout layer is trialed between the 2 hidden layers to randomly set a fraction of input units to 0 to prevent overfitting.

To optimize model hyperparameters, we wrote for loops to try different combinations of:

- Batch Sizes - [50,100]
- Number of Epochs - [100,200,250]
- Layer 1 widths - [10,20,30,40,50,60]
- Layer 2 widths - [5,10,20,50]
- Dropout levels - [0,0.2,0.5]

For each combination, we plotted loss, accuracy, precision and recall for both training and validation data. To choose the best model, we considered two factors:

- Overfitting
- Accuracy, Precision and Recall metrics for validation data set
    - As there is a data imbalance in the class distribution (Figure 5.1), we need to look beyond accuracy as just predicting a single class can lead to high accuracy (~85% in our case). Therefore, we need to look at precision and recall.

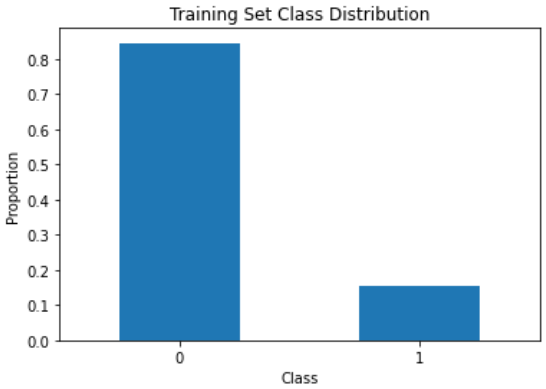

Figure 5.1 Class distribution of training, validation, and test set

- Approving a bad loan is more expensive than missing a potential customer so prioritised high recall (% of actual bad loans identified) to capture as many bad loans as possible with the potential trade-off of a lower precision (% of correct bad loan predictions)

Optimal conditions were found with:

- Layer 1 Width=50
- Layer 2 Width=5
- Batch-size=50
- Epochs=200
- Dropout=0.5

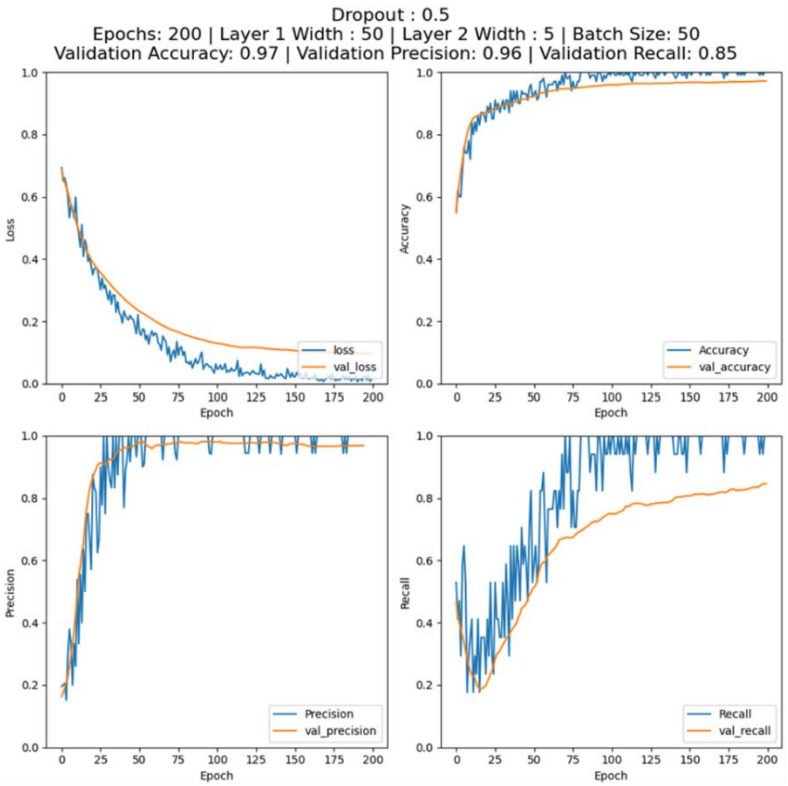

Figure 5.2. Loss, Accuracy, Precision, and Recall performance for training and validation data

Looking closer, when we deviate from optimal solutions, we observe worse performance:

- Increasing batch size from 50 to 100 results in poor recall and a decreasing precision (Figure A.4.1)
- Fewer epochs (100) lead to poorer metrics. More epochs (250) cause overfitting as seen by the deviation in training and validation performance (Figure A.4.2)
- A Lower dropout rate (0.2) or no dropout increases the overfitting as seen by the deviation in training and validation performance (Figure A.4.3)
- A wider (60) or narrower (40) layer 1 also affects performance. In both cases we see a greater degree of overfitting as well as worse metrics (Figure A.4.4)
- Increasing layer 2 width to 10 increases the amount of deviation in model performance implying overfitting (Figure A.4.5)

To address class imbalance, we re-ran model after performing SMOTE as it can be beneficial for DNN models to oversample the minority class (Dablain, 2022). However, we found more signs of overfitting and poorer metrics (Figure A.4.6). There was greater deviation of validation performance from training and precision dropped after a certain number of epochs, possibly due to memorisation of training data rather than generalising for data.

We use the optimal model without any oversampling on the data on the test data for evaluation.

6. Evaluation

DL model results were evaluated using various techniques:

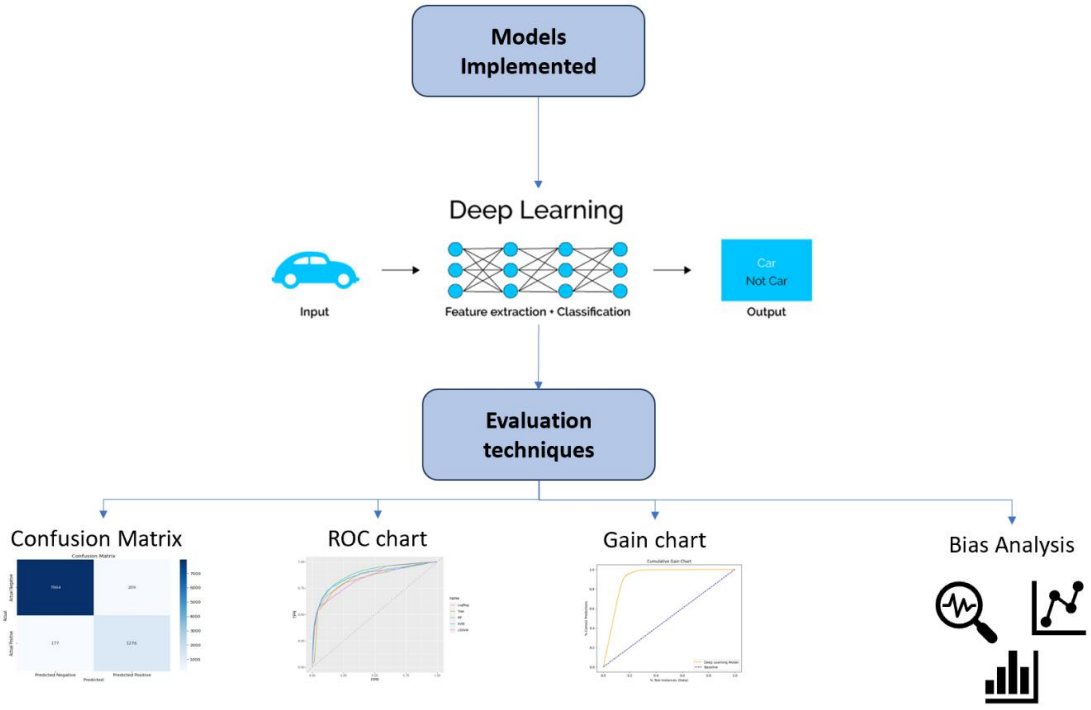

Figure 6.1: Flowchart illustrates the model evaluation techniques

a. Confusion Matrix

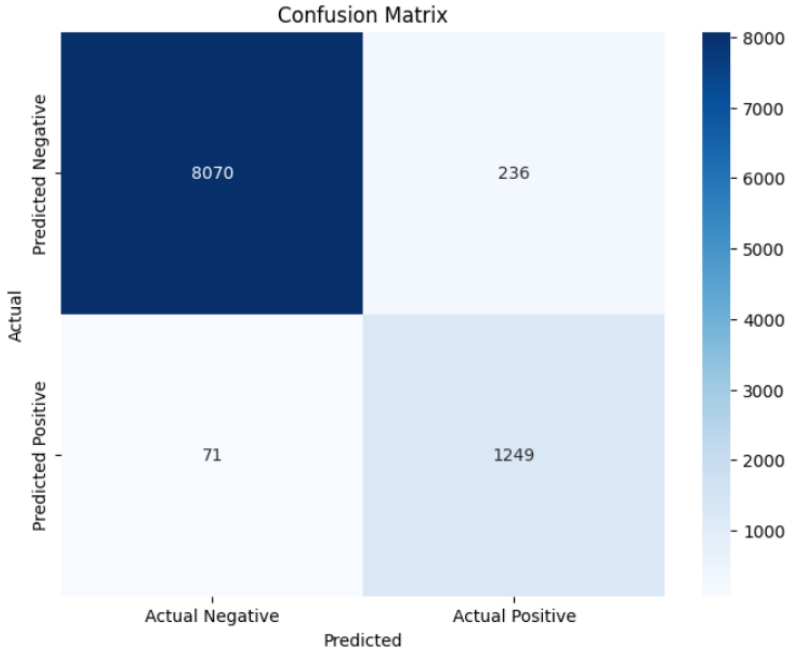

Figure 6.2. Confusion Matrix

From Figure 6.3, the model has 96.8% accuracy to predict most of loan outcomes in test dataset. 84% (Recall) of the actual bad loan applications are identified by model. 94.62% (Precision) of the predicted bad loan predictions are actual bad loans. High F-1 Score (89%) denotes a good balance between precision and recall. These measures indicate company losses can potentially be reduced by identifying significant numbers of bad loans.

Measure	Result(in percentage)
Accuracy	96.81
Precision	94.62
Recall	84.1
F1-Measure	89.05

Figure 6.3. Results of evaluation metrics

b. AUC and ROC chart

From Figure 6.4, our model's ROC curve is very close to the top left corner indicating higher sensitivity (or TPR) and low false positive mistakes across various threshold settings. AUC of 0.98 is very close to 1, indicating strong ability of our model to discriminate between classes.

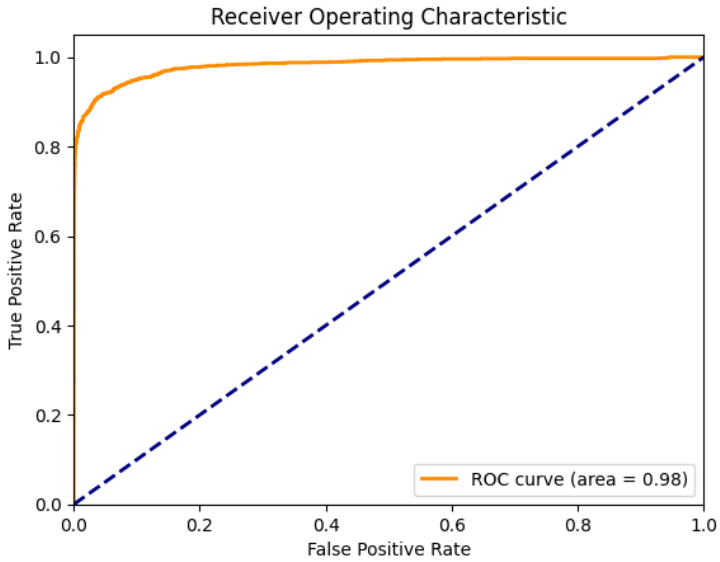

Figure 6.4. Receiver Operator Characteristic (ROC) Graph

c. Gain chart

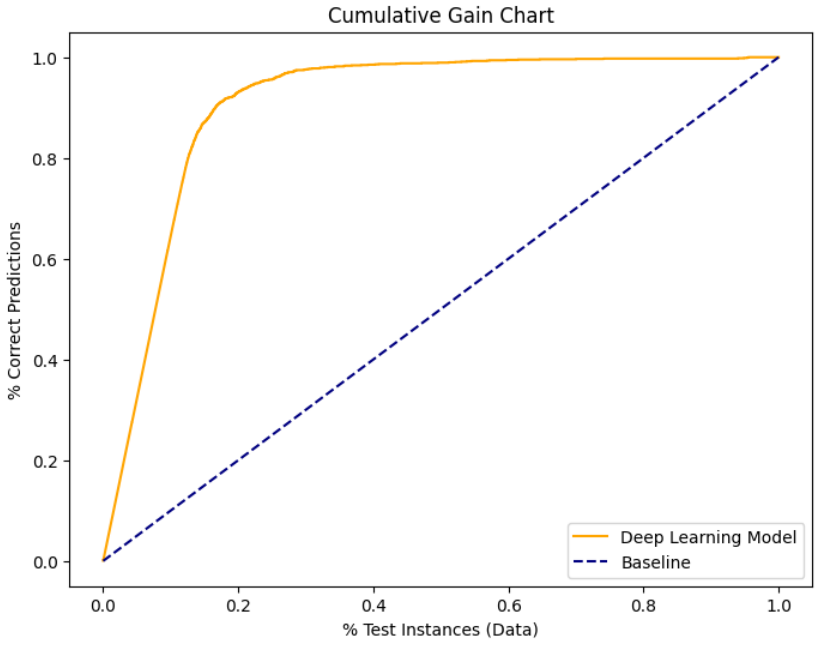

Figure 6.5. Cumulative gain graph

The gain curve is quite steep, demonstrating that our model is efficient in prioritizing instances of bad loans. Specifically, targeting just 20% of test instances could capture ~90% of all possible bad loans.

d. Bias analysis

i. Income levels

Income Levels	Percentage of bad loan predictions by the model	Percentage of actual bad loans	Difference in Percentage
High	8.61	12.15	3.54
Medium	12.18	16.05	3.87
Low	11.82	17.96	6.14

Figure 6.6. Bias Analysis based on income levels

From Figure 6.6, the percentage of actual bad loans in low-income levels is much higher when compared to the number of bad loans predicted by the model, showing model bias towards low-income levels compared to other levels.

ii. Loan grade

Loan Grade	Percentage of bad loan predictions by the model	Percentage of actual bad loans	Difference in Percentage
A	3.99	6.35	2.36
B	7.71	11.84	4.13
C	12.3	17.72	5.42
D	24.35	28.23	3.88
E	16.34	22.79	6.45
F	25.21	29.73	4.52
G	31.92	40.85	8.93

Figure 6.7. Bias Analysis based on Loan Grade

Based on Figure 6.7, model bias towards loan applicants in loan grades 5 and 7 is higher than other loan grade levels due to the high percentage difference between the bad loan predictions and actual bad loans.

iii. Region

Region	Percentage of bad loan predictions by the model	Percentage of actual bad loans	Difference in Percentage
Northeast	11.78	16.65	4.87
South	12.09	15.44	3.35
West	9.5	15.11	5.61
Mid-West	4.28	7.45	3.17

Figure 6.8. Bias Analysis based on region

Results of the bias analysis done region-wise show that the bias error rate is significantly higher for West region. The model performs poorly in predicting bad loans from this region.

7. Limitation

- **Data Quality Problem**

DNN model typically requires large amounts of high-quality data for effective training. In our case, there are many missing values in credit history columns, and we need to handle these missing values effectively to improve the accuracy and predictive performance of the model.

- **Interpretability of Model**

DL model's interpretability remains a concern as it poses potential risks (including regulatory issues) due to its black-box approach to learning (Liang et al., 2021). In the financial sector, transparency and explainability in the decision-making process are considered important. With DL models it is difficult to understand which features contribute to the predictions, making these features opaquer.

- **Difficulty in reproduction**

Significant randomness exists in training D models, including random initialization of parameters, random optimization algorithms. This randomness would lead to variations in results even with identical settings.

- **Computation Intensive**

Training DNN requires a lot of computational resources, especially as the network size grows (Alexandropoulos et al., 2019). Computation can become a bottleneck for large scale problems.

8. Recommendation

Based on the results, DNN model performs quite well in predicting who will default ('loan_is_bad') and the company can use it as an automated solution to loan applications. If the company's goal is to correctly identify as many defaulters as possible and prevent the future risks of incorrectly identifying default cases, the company should train the model to maximize recall. Conversely, if the cost of refusing a good loan is high, it should train the model to maximize precision. If the company wants to balance both then it can prioritize F1-score. However, our model has a bias towards the group of people with low income, loan grades of E and G and living in the West region.

Due to these limitations, we recommend employing advanced data imputation for missing values and enhance interpretability through techniques like SHAP and LIME. Ensuring reproducibility requires strict documentation and deterministic settings, while computational demands can be mitigated by simplifying models, leveraging cloud computing resources, and applying distributed training. Together these strategies can improve the robustness, transparency, and efficiency of DNNs in complex financial applications, aligning with both performance goals and regulatory standards.

9. Conclusion

This study explored DL's effectiveness in predicting loan defaults for our consumer lending company. We reached reasonable recommendations based on our findings and identified areas for future research. DNN achieved significant success in predicting defaults, demonstrating its robustness through strong metrics. We ensured a conservative approach to credit risk management minimizing potential losses by achieving a high recall rate. Additionally, DL models could offer a more scalable approach than traditional methods, potentially streamlining operations and improving overall performance. Some limitations of this approach could rely on data quality issues, model interpretability, inherent DL reproducibility difficulties, computation intensive. Future research directions could involve investigating ensemble learning methods, incorporating explainable AI techniques, and exploring alternative DL architectures tailored to specific lending contexts.

In conclusion, adopting a well-structured DL model could be a game-changer for the company and this innovative approach could lead to a competitive advantage and sustainable business growth.

References

Abakarim, Y., Lahby, M. and Attioui, A. (2018). 'Towards an efficient real-time approach to loan credit approval using deep learning'. In 2018 9th International Symposium on Signal, Image, Video and Communications (ISIVC) (pp. 306-313). IEEE.

Alberto Landi et al. (2010). 'Artificial neural networks for nonlinear regression and classification'. Intelligent Systems Design and Applications (ISDA) 2010 10th International Conference on.

Alexandropoulos, S.-A.N. *et al.* (2019) 'A deep dense neural network for bankruptcy prediction', *Engineering Applications of Neural Networks*, pp. 435–444. doi:10.1007/978-3-030-20257-6_37.

Bayraci, S. and Susuz, O. (2019). 'A Deep Neural Network (DNN) based classification model in application to loan default prediction'. Theoretical and Applied Economics, 4(621), pp.75-84.

Byanjankar, A., Heikkilä, M. and Mezei, J. (2015). 'Predicting credit risk in peer-to-peer lending: A neural network approach'. 2015 IEEE symposium series on computational intelligence (pp. 719-725). IEEE.

Dablain, D., Krawczyk, B. and Chawla, N.V. (2022). DeepSMOTE: Fusing Deep Learning and SMOTE for Imbalanced Data. *IEEE Transactions on Neural Networks and Learning Systems*, pp.1–15

De Boer, P.-T., Kroese, D. P., Mannor, S., & Rubinstein, R. Y. (2005). A Tutorial on the Cross-Entropy Method. Annals of Operations Research, 134, 19–67

François Chollet, Keras: Deep learning library for theano and tensorflow., vol. 7, no. 8, 2015, [online] Available at: https://keras.io/k.

Hamori, S., Kawai, M., Kume, T., Murakami, Y. and Watanabe, C. (2018). 'Ensemble Learning or Deep Learning? Application to Default Risk Analysis'. Journal of Risk and Financial Management, 11(1), pp. 12.

Hastie, T., Tibshirani, R., & Friedman, J. (2009). The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer Series in Statistics.

Herbert L. JENSEN. (1992), 'Using neural networks for credit scoring'. Managerial finance, vol. 18, no. 6, pp. 15-26.

Kang FU, Dawei CHENG, Yi TU et al. (2016). 'Credit Card Fraud Detection Using Convolutional Neural Networks'. International Conference on Neural Information Processing, pp. 483-490.

Kvamme, H., Sellereite, N., Aas, K. and Sjursen, S. (2018). 'Predicting mortgage default using convolutional neural networks'. Expert Systems with Applications, 102, pp. 207-217.

Kumar, M., Goel, V., Jain, T., Singhal, S. and Goel, L. (2018). 'Neural network approach to loan default prediction'. International Research Journal of Engineering and Technology (IRJET), 5(4), pp.4-7.

Liang, Y., Li, S., Yan, C., Li, M. and Jiang, C. (2021). 'Explaining the black-box model: A survey of local interpretation methods for deep neural networks'. Neurocomputing, 419, pp.168-182.

Liu, M., Li, S., Yuan, H., Eng, M., Ning, Y., Xie, F.(2023). 'Handling missing values in healthcare data: A systematic review of deep learning-based imputation techniques'. 142, pp.102587–102587. doi:https://doi.org/10.1016/j.artmed.2023.102587.

Lu, Z., Pu, H., Wang, F., Hu, Z. and Wang, L. (2010). The Expressive Power of Neural Networks: A View from the Width. [online] Available at: https://proceedings.neurips.cc/paper_files/paper/2017/file/32cbf687880eb1674a07bf717761dd3a-Paper.pdf

Nair, V., & Hinton, G. E. (2010). Rectified Linear Units Improve Restricted Boltzmann Machines. In Proceedings of the 27th International Conference on Machine Learning (ICML-10)

Nazari, F. and Yan, W. (n.d.). Convolutional versus Dense Neural Networks: Comparing the Two Neural Networks' Performance in Predicting Building Operational Energy Use Based on the Building Shape. Available at: https://arxiv.org/pdf/2108.12929.pdf.

Pang, B., Nijkamp, E. and Wu, Y.N. (2020). 'Deep learning with tensorflow: A review'. Journal of Educational and Behavioral Statistics, 45(2), pp.227-248.

Y. Bengio. (2009). 'Learning deep architectures for ai'. Found. Trends Mach. Learn., vol. 2, no. 1, pp. 1-127.

Appendix

A. Python Deep Learning code

```
# Import all the necessary libraries
import tensorflow as tf
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix,precision_score, recall_score,roc_curve,
auc,roc_auc_score
import matplotlib.pyplot as plt
from keras import losses
import seaborn as sns

# Reading the file

loan_data = pd.read_excel('loan_data_ADA_assignment.xlsx')

# PART 1: DATA PREPARATION

# Check the structure of data set
loan_data.info()

# Identifying duplicate rows. There is no duplicate row in loan data set
num_duplicate_rows = loan_data.duplicated().sum()
print("Number of duplicate rows:", num_duplicate_rows)

# Checking for missing values
total_na = loan_data.isna().sum().sum()
print(total_na)
```

```
# Checking for missing values by column
na_counts_per_column = loan_data.isna().sum()
print(na_counts_per_column)

# Converting Mnths since columns to binary
loan_data['delinquency'] = np.where(loan_data['mths_since_last_delinq'].notna(), 1, 0)
loan_data['derogatory'] = np.where(loan_data['mths_since_last_major_derog'].notna(), 1, 0)
loan_data['record'] = np.where(loan_data['mths_since_last_record'].notna(), 1, 0)

# Get unique addr_state names
unique_addr_state = loan_data['addr_state'].unique()

# Print the list of unique_addr_state
print(unique_addr_state)

# Define regions
regions = {
    'Northeast': ['ME', 'NH', 'VT', 'MA', 'RI', 'CT', 'NY', 'NJ', 'PA'],
    'Midwest': ['OH', 'MI', 'IN', 'WI', 'IL', 'MN', 'IA', 'MO', 'ND', 'SD', 'NE', 'KS'],
    'South': ['DE', 'MD', 'DC', 'VA', 'WV', 'NC', 'SC', 'GA', 'FL', 'KY', 'TN', 'MS', 'AL', 'OK', 'TX',
'AR', 'LA'],
    'West': ['ID', 'MT', 'WY', 'CO', 'NM', 'AZ', 'UT', 'NV', 'CA', 'OR', 'WA', 'AK', 'HI']}

# Mapping function from state to region
def map_state_to_region(state):
    for region, states in regions.items():
        if state in states:
            return region
```

```
# Adding a new 'Region' column to the loan_data DataFrame
loan_data['region'] = loan_data['addr_state'].apply(map_state_to_region)

# Check value in region column, which is including 'South','Northeast','Midwest','West'
unique_regions = loan_data['region'].unique()
print(unique_regions)

#Dropping noise columns, High NA values and columns not suitable for one-hot encoding)
columns_to_drop = ["next_pymnt_d", "mths_since_last_delinq",
"mths_since_last_major_derog",
                   "mths_since_last_record", "tot_coll_amt", "tot_cur_bal", "total_credit_rv",
                   "id", "member_id", "emp_title", "pymnt_plan", "desc", "title",
                   "policy_code", "zip_code", "addr_state"]

loan_data.drop(columns=columns_to_drop, inplace=True)

# Checking for missing values again
total_na = loan_data.isna().sum().sum()
print(total_na)

# Since the number of missing values are less than 5% of the dataset, we can remove them.
loan_data.dropna(inplace=True)

# Converting categorical variables to 'category' type
loan_data['grade'] = loan_data['grade'].astype('category')
loan_data['sub_grade'] = loan_data['sub_grade'].astype('category')

## Label encoding
# Ensure grade, home_ownership, verification_status are category with the specified levels
grade_categories = ["A", "B", "C", "D", "E", "F", "G"]
```

```
# Convert the category to ordered categories
loan_data['grade'] = pd.Categorical(loan_data['grade'], categories=grade_categories,
ordered=True)

# Replace categories with their numerical codes
loan_data['grade'] = loan_data['grade'].cat.codes+1

# sub_grade
# Create an ordered category with levels
loan_data['sub_grade'] = pd.Categorical(loan_data['sub_grade'], ordered=True)
loan_data['sub_grade'] = loan_data['sub_grade'].cat.codes+1

## One-Hot encoding on 'purpose','loan_status','home_ownership','verification_status','region'
loan_data = pd.get_dummies(loan_data, columns=['purpose', 'loan_status',
                                    'home_ownership','verification_status','region'
                                    ], drop_first=True, dtype=int)

# Converting dates into days from 1st January 2024
dates = ['issue_d', 'last_pymnt_d', 'earliest_cr_line', 'last_credit_pull_d']

for column in dates:
    # Convert month-year strings to datetime, assuming the first day of the month
    loan_data[column] = pd.to_datetime(loan_data[column], format='%b-%y', errors='coerce')

    # Calculate the number of days from a reference date
    loan_data[column] = (pd.Timestamp('2024-01-01') - loan_data[column]).dt.days

# Standardising the Data
scaler = StandardScaler()
```

```
loan_data_scaled = scaler.fit_transform(loan_data)
data = pd.DataFrame(loan_data_scaled, columns=loan_data.columns,index=loan_data.index)
data['loan_is_bad'] = loan_data['loan_is_bad']

# Converting the loan_is_bad column to binary
data['loan_is_bad'] = loan_data['loan_is_bad'].astype(int)

# Check the value of Target variable
print(data['loan_is_bad'].head())

#Show bit of data
data.head()

#Separate the target column from the data
var_target = data.pop('loan_is_bad')

# Check data size again
print(len(data))

# PART 2: DATA PARTITIONING

#Split the data set into 80% training and 20% test set
from sklearn.model_selection import train_test_split
train_val_data, test_data, train_val_labels, test_labels = train_test_split(data,
            var_target,
            test_size=0.2,
            stratify=var_target,
            random_state=42)

# Split the training and validation set into 70% training and 10% validation
```

```
from sklearn.model_selection import train_test_split
train_data, val_data, train_labels, val_labels = train_test_split(train_val_data,
        train_val_labels,
        test_size=0.125,  # 0.125 x 0.8 = 0.1
        stratify=train_val_labels,
        random_state=42)

# Check the class distribution in the target column for loan_data_scaled, training, validation and test data set

# Data dataset
class_distribution = var_target.value_counts()
print("Training Set Class Distribution:\n", class_distribution)

class_distribution_percetage = var_target.value_counts(normalize=True) * 100
print("Class distribution in the data dataset:")
print(class_distribution_percetage)

# Training data set
# For absolute counts
train_class_distribution = train_labels.value_counts()
print("Training Set Class Distribution:\n", train_class_distribution)

# For percentages
train_class_distribution_percent = train_labels.value_counts(normalize=True)
print("Training Set Class Distribution Percentage:\n", train_class_distribution_percent)

# Validation data set
# For absolute counts
```

```
val_class_distribution = val_labels.value_counts()
print("Validation Set Class Distribution:\n", val_class_distribution)

# For percentages
val_class_distribution_percent = val_labels.value_counts(normalize=True)
print("Validation Set Class Distribution Percentage:\n", val_class_distribution_percent)

# Test data set
# For absolute counts
test_class_distribution = test_labels.value_counts()
print("Test Set Class Distribution:\n", test_class_distribution)

# For percentages
test_class_distribution_percent = test_labels.value_counts(normalize=True)
print("Test Set Class Distribution Percentage:\n", test_class_distribution_percent)

# Visualizing Class Distribution
def plot_class_distribution(labels, title):
    labels.value_counts(normalize=True).plot(kind='bar')
    plt.title(title)
    plt.xlabel('Class')
    plt.ylabel('Proportion')
    plt.xticks(rotation=0)  # Keeps the class labels horizontal
    plt.show()

plot_class_distribution(train_labels, 'Training Set Class Distribution')
plot_class_distribution(val_labels, 'Validation Set Class Distribution')
plot_class_distribution(test_labels, 'Test Set Class Distribution')

# Use oversampling to deal with data imbalance
```

```
#from imblearn.over_sampling import SMOTE

# Initialize SMOTE
#smote = SMOTE(random_state=42)

# Apply SMOTE to the training data only
#train_data, train_labels = smote.fit_resample(train_data, train_labels)

# Verify the class distribution is now 50/50
#print("After SMOTE - Counts of label '1':", sum(train_labels == 1))
#print("After SMOTE - Counts of label '0':", sum(train_labels == 0))

#print("\nClass distribution after SMOTE:")
#print(train_labels.value_counts(normalize=True))

# PART 3: DEEP LEARNING MODEL
# Convert the training and test sets to TensorFlow datasets
train_dataset = tf.data.Dataset.from_tensor_slices((train_data.values, train_labels.values))
val_dataset = tf.data.Dataset.from_tensor_slices((val_data.values, val_labels.values))
test_dataset = tf.data.Dataset.from_tensor_slices((test_data.values, test_labels.values))

#Loops to optimise hyperparameters and generate training vs validation metric plots
# Define different parameters to iterate over
optimizers = [tf.keras.optimizers.AdamW]
loss_functions = [tf.keras.losses.BinaryCrossentropy]
metrics = [['accuracy', 'Precision', 'Recall']]
epochs_range = [100,200,250]  # Max 100 epochs
layer_width1 = [10,20,30,40,50,60]
layer_width2 = [5,10,20,30]
```

```
dropouts = [0,0.2,0.5]
batch_size = [50,100]


# This part we have run to check different combination of parameters and it is working but it takes long time to work
# # Iterate over parameter combinations
# for epochs in epochs_range:
#     for l1 in layer_width1:
#             for l2 in layer_width2:
#                     for batch_size in batch_size:
#                         for dropout_rate in dropouts:
#                             # Create modelRR
#                             #set batch size
#                             train_batch = train_dataset.batch(batch_size) # batch size = 50
#                             features, labels = next(iter(train_batch)) # iterate through each batch at training time
#                             model = tf.keras.Sequential([
#                                 tf.keras.layers.Dense(l1, activation=tf.nn.relu, input_shape=(len(train_data.columns), )),
#                                 tf.keras.layers.Dense(l2, activation=tf.nn.relu),
#                                 tf.keras.layers.Dropout(dropout_rate),
#                                 tf.keras.layers.Dense(1, activation='sigmoid')
#                                 ])


#                             # Compile model with selected optimizer, loss function, and metrics
#                             model.compile(optimizer=tf.keras.optimizers.AdamW(learning_rate=0.001, weight_decay=0.005),
#                                           loss=tf.keras.losses.BinaryCrossentropy(),
#                                           metrics=['accuracy', 'Precision', 'Recall'])


#                             # Train the model
```

```
#                   historycheck = model.fit(features, labels,
#                                       epochs=epochs,
#                                       batch_size=batch_size,
#                                       validation_data=(val_data, val_labels),
#                                       verbose=0)  # Set verbose=0 to suppress output
#                   # Create a new figure
#                   fig = plt.figure(figsize=(10, 10))


#                   # Add overall title


#                   fig.suptitle(f"Dropout : {dropout_rate} \n  Epochs: {epochs} | Layer 1
Width : {l1} | Layer 2 Width : {l2} | Batch Size: {batch_size} \n Validation Accuracy:
{round(historycheck.history['val_accuracy'][-1],2)} | Validation Precision:
{round(historycheck.history['val_Precision'][-1],2)} | Validation Recall:
{round(historycheck.history['val_Recall'][-1],2)}" , fontsize=16)


#                   # Subplot for loss
#                   plt.subplot(2, 2, 1)
#                   plt.plot(historycheck.history['loss'], label='loss')
#                   plt.plot(historycheck.history['val_loss'], label='val_loss')
#                   plt.xlabel('Epoch')
#                   plt.ylabel('Loss')
#                   plt.ylim([0, 1])
#                   plt.legend(loc='lower right')


#                   # Subplot for accuracy
#                   plt.subplot(2, 2, 2)
#                   plt.plot(historycheck.history['accuracy'], label='Accuracy')
#                   plt.plot(historycheck.history['val_accuracy'], label='val_accuracy')
#                   plt.xlabel('Epoch')
#                   plt.ylabel('Accuracy')
```

```
#                   plt.ylim([0, 1])
#                   plt.legend(loc='lower right')

#                   # Subplot for precision
#                   plt.subplot(2, 2, 3)
#                   plt.plot(historycheck.history['Precision'], label='Precision')
#                   plt.plot(historycheck.history['val_Precision'], label='val_precision')
#                   plt.xlabel('Epoch')
#                   plt.ylabel('Precision')
#                   plt.ylim([0, 1])
#                   plt.legend(loc='lower right')

#                   # Subplot for recall
#                   plt.subplot(2, 2, 4)
#                   plt.plot(historycheck.history['Recall'], label='Recall')
#                   plt.plot(historycheck.history['val_Recall'], label='val_recall')
#                   plt.xlabel('Epoch')
#                   plt.ylabel('Recall')
#                   plt.ylim([0, 1])
#                   plt.legend(loc='lower right')

#                   # Show the plot
#                   plt.tight_layout()
#                   plt.subplots_adjust(top=0.9)
#                   plt.show()

### Setting optimal parameters
batch_size = 50
epochs=200
dropout = 0.5
```

```
layer1_width = 50
layer2_width = 5

#Set batch size for training dataset
train_batch = train_dataset.batch(batch_size)
features, labels = next(iter(train_batch))
 # iterate through each batch at training time

#set batch size
train_batch = val_dataset.batch(len(val_dataset)) # batch size = 50
val_features, val_labels = next(iter(train_batch))

#Input layer, width and depth for the model
model = tf.keras.Sequential([
  tf.keras.layers.Dense(layer1_width, activation=tf.nn.relu,
input_shape=(len(train_data.columns), )),
  tf.keras.layers.Dropout(dropout),
  tf.keras.layers.Dense(layer2_width, activation=tf.nn.relu), ## copy and paste below to add layer
  tf.keras.layers.Dense(1, activation = 'sigmoid')
])

#Setting optimiser, loss function and metrics for the model
model.compile(optimizer=tf.keras.optimizers.AdamW(learning_rate=0.001,
weight_decay=0.005),
              loss=tf.keras.losses.BinaryCrossentropy(), ### loss function
              metrics=(['Precision','Recall', 'accuracy'])) ### metric

#Fitting the model
historycheck = model.fit(features, labels, epochs=epochs, validation_data=(val_data, val_labels))
```

```
#Plot graphs for training vs validation metrics for loss, accuracy, precision and recall

# Create a new figure

fig = plt.figure(figsize=(10, 10))


# Add overall title


fig.suptitle(f"Dropout : 0.5 \n  Epochs: {epochs} | Layer 1 Width : {layer1_width} | Layer 2 
Width : {layer2_width} | Batch Size: {batch_size} \n Validation Accuracy: 
{round(historycheck.history['val_accuracy'][-1],2)} | Validation Precision: 
{round(historycheck.history['val_Precision'][-1],2)} | Validation Recall: 
{round(historycheck.history['val_Recall'][-1],2)}" , fontsize=16)


# Subplot for loss

plt.subplot(2, 2, 1)

plt.plot(historycheck.history['loss'], label='loss')

plt.plot(historycheck.history['val_loss'], label='val_loss')

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.ylim([0, 1])

plt.legend(loc='lower right')


# Subplot for accuracy

plt.subplot(2, 2, 2)

plt.plot(historycheck.history['accuracy'], label='Accuracy')

plt.plot(historycheck.history['val_accuracy'], label='val_accuracy')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.ylim([0, 1])

plt.legend(loc='lower right')


# Subplot for precision
```

```
plt.subplot(2, 2, 3)
plt.plot(historycheck.history['Precision'], label='Precision')
plt.plot(historycheck.history['val_Precision'], label='val_Precision')
plt.xlabel('Epoch')
plt.ylabel('Precision')
plt.ylim([0, 1])
plt.legend(loc='lower right')

# Subplot for recall
plt.subplot(2, 2, 4)
plt.plot(historycheck.history['Recall'], label='Recall')
plt.plot(historycheck.history['val_Recall'], label='val_Recall')
plt.xlabel('Epoch')
plt.ylabel('Recall')
plt.ylim([0, 1])
plt.legend(loc='lower right')

# Show the plot
plt.tight_layout()
plt.subplots_adjust(top=0.9)
plt.show()

# PART 4: EVALUATION PART
#feeding the entire test data into the model at once
test_batch = test_dataset.batch(len(test_data)) # -- the size of the the whole dataset
test_features, test_labels = next(iter(test_batch))

#Evaluate test metrics
test_loss, test1, test2, test3 = model.evaluate(test_features,  test_labels, verbose=2)
```

```
#Original number of positives
#sum(test_labels)
#Predictons
test_predictions = model.predict(test_features)
##converting probability predictions into binary
test_binary_predictions = [1 if pred >= 0.5 else 0 for pred in test_predictions]



#Create confusion matrix
confusion_test = confusion_matrix(test_binary_predictions,test_labels)

test_TN,test_FN, test_FP, test_TP = confusion_test.ravel()

#confusion_test

# Check for element in confusion matrix
test_FN

# Visualise confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(confusion_test, annot=True, fmt="d", cmap="Blues",
        yticklabels=['Predicted Negative', 'Predicted Positive'],
        xticklabels=['Actual Negative', 'Actual Positive'])
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.title('Confusion Matrix')
plt.show()

# Calculate metrics
test_accuracy = (test_TP + test_TN) / (test_TP + test_TN + test_FP + test_FN)
```

```
test_precision = test_TP / (test_TP + test_FP)
test_recall = test_TP / (test_TP + test_FN)
test_f1_score = 2 * (test_precision * test_recall) / (test_precision + test_recall)

# Print metrics
print(f"test_accuracy: {test_accuracy}")
print(f"test_precision: {test_precision}")
print(f"test_recall: {test_recall}")
print(f"test_F1 Score: {test_f1_score}")

## combine test dataset + original target variable + prediction variables
test_data_results=pd.DataFrame(test_features.numpy())
test_data_results['loan_is_bad'] = test_labels  # add original_target column
test_data_results['predictions'] = test_binary_predictions  # add predictions column
test_data_results['Correct_Prediction'] = np.where(test_data_results['loan_is_bad'] ==
test_data_results['predictions'], 1, 0)

# ROC chart
# Compute ROC curve
fpr, tpr, thresholds = roc_curve(test_labels, model.predict(test_features))

# Compute ROC area under the curve
roc_auc = auc(fpr, tpr)

# Plot ROC curve
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
```

```
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")
plt.show()

# First, get the probabilities of the positive class
predicted_probs = model.predict(test_features).ravel()  # Flatten to 1D

# Calculate the AUC
auc_score = roc_auc_score(test_labels, predicted_probs)
print(f"AUC: {auc_score}")

# Generate an array of percentages to represent the x-axis from 0 to 100
percentages = np.linspace(0, 1, len(predicted_probs))

# Sort the probabilities and the corresponding labels
sorted_indices = np.argsort(predicted_probs)[::-1]
sorted_labels = np.array(test_labels)[sorted_indices]

# Calculate the cumulative gain
cumulative_gain = np.cumsum(sorted_labels) / np.sum(sorted_labels)

# Calculate the baseline (random model)
baseline = np.linspace(0, 1, len(cumulative_gain))

# Plot the Gain Chart
plt.figure(figsize=(8, 6))
plt.plot(percentages, cumulative_gain, label='Deep Learning Model', color='orange')
plt.plot(percentages, baseline, label='Baseline', linestyle='--', color='navy')
```

```
plt.xlabel('% Test Instances (Data)')
plt.ylabel('% Correct Predictions')
plt.title('Cumulative Gain Chart')
plt.legend(loc='lower right')
plt.show()


#full income test
# Convert the training and test sets to TensorFfull datasets
full_test_dataset = tf.data.Dataset.from_tensor_slices((data.values, var_target.values))
#feeding the entire test data into the model
test_batch = full_test_dataset.batch(len(full_test_dataset)) # -- the size of the the whole dataset
full_features, full_labels = next(iter(test_batch))


## this is the metrics that we used earlier for the model. For example right now, test1 is precision and test2 is recall. if you change the number of metrics you have to change the number of tests here as well can be called anything
full_test_loss, full_test1, full_test2, full_test3 = model.evaluate(full_features,  full_labels, verbose=2)


#Predictons
full_predictions = model.predict(full_features)
##converting probability predictions into binary
full_binary_predictions = [1 if pred >= 0.5 else 0 for pred in full_predictions]


## combine full dataset + original target variable + prediction variables
full_data_results=loan_data
full_data_results['predictions'] = full_binary_predictions  # add predictions column


# Create 'Predicted Correctly' column
full_data_results['Correct_Prediction'] = np.where(full_data_results['loan_is_bad'] == full_data_results['predictions'], 1, 0)
```

```
full_data_results.head()


#Income level generation

full_data_results['income_level'] = pd.qcut(full_data_results['annual_inc'], q=3, labels=['low',
'medium', 'high'])


# Bias analysis


# Create 'Predicted Correctly' column

full_data_results['Correct_Prediction'] = np.where(full_data_results['loan_is_bad'] ==
full_data_results['predictions'], 1, 0)


full_data_results.head()


full_data_results['income_level'] = pd.qcut(full_data_results['annual_inc'], q=3, labels=['low',
'medium', 'high'])



#Bias Analysis on income levels in the dataset

for i in loan_data['income_level'].unique():

    n_predictions_dl=len(full_data_results[(full_data_results['Correct_Prediction']==1) &
(full_data_results['predictions']==1) & (full_data_results['income_level']==i)])

    n_actual= len(full_data_results[(full_data_results['loan_is_bad']==1) &
(full_data_results['income_level']==i)])

    total= len(full_data_results[(full_data_results['income_level']==i)])

    print("Percentage of predictions of bad loans by model in",i ,'income
category:',round((n_predictions_dl/total)*100,2))

    print("Percentage of actual bad loans in",i ,'income category:',round((n_actual/total)*100,2))


#Bias check in loan grade

for i in loan_data['grade'].unique():
```

```
    n_predictions_dl = len(full_data_results[((full_data_results['Correct_Prediction']==1) &
full_data_results['predictions']==1) & (full_data_results['grade']==i)])

    n_actual= len(full_data_results[(full_data_results['loan_is_bad']==1) &
(full_data_results['grade']==i)])

    total= len(full_data_results[(full_data_results['grade']==i)])

    print("Percentage of bad loan predictions by the model in loan
grade",i ,":",round((n_predictions_dl/total)*100,2))

    print("Percentage of actual bad loans in loan grade",i,":",round((n_actual/total)*100,2))


#Bias check in Northeast region


n_predictions_dl=len(full_data_results[(full_data_results['Correct_Prediction']==1) &
(full_data_results['predictions']==1) & (full_data_results['region_Northeast']==1)])

n_actual= len(full_data_results[(full_data_results['loan_is_bad']==1) &
(full_data_results['region_Northeast']==1)])

total= len(full_data_results[(full_data_results['region_Northeast']==1)])

print("Percentage of predictions of bad loans by model in Northeast
region",round((n_predictions_dl/total)*100,2))

print("Percentage of actual bad loans in  Northeast region",round((n_actual/total)*100,2))


#Bias check in South region


n_predictions_dl=len(full_data_results[(full_data_results['Correct_Prediction']==1) &
(full_data_results['predictions']==1) & (full_data_results['region_South']==1)])

n_actual= len(full_data_results[(full_data_results['loan_is_bad']==1) &
(full_data_results['region_South']==1)])

total= len(full_data_results[(full_data_results['region_South']==1)])

print("Percentage of predictions of bad loans by model in South
region",round((n_predictions_dl/total)*100,2))

print("Percentage of actual bad loans in  South region",round((n_actual/total)*100,2))


#Bias check in West region
```

```
n_predictions_dl=len(loan_data[(full_data_results['Correct_Prediction']==1) &
(full_data_results['predictions']==1) & (full_data_results['region_West']==1)])

n_actual= len(loan_data[(full_data_results['loan_is_bad']==1) &
(full_data_results['region_West']==1)])

total= len(full_data_results[(full_data_results['region_West']==1)])

print("Percentage of predictions of bad loans by model in West
region",round((n_predictions_dl/total)*100,2))

print("Percentage of actual bad loans in  West region",round((n_actual/total)*100,2))
```

A.4. Graphs

Figures:

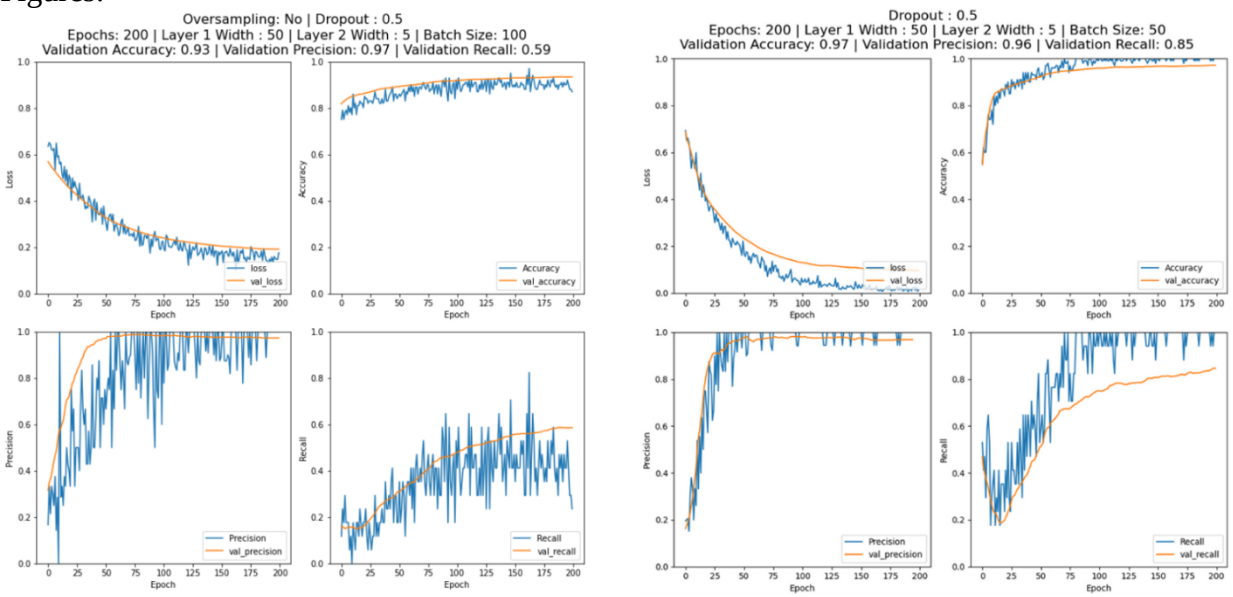

Figure A.4.1 Models tested with different batch sizes (100 and 50)

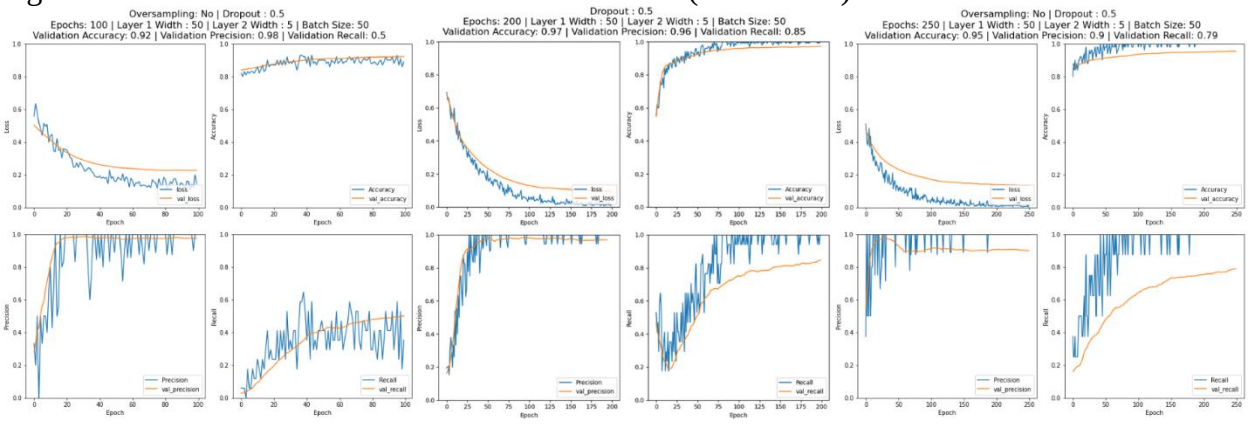

Figure A.4.2 Models tested with different epochs (100, 200 and 250)

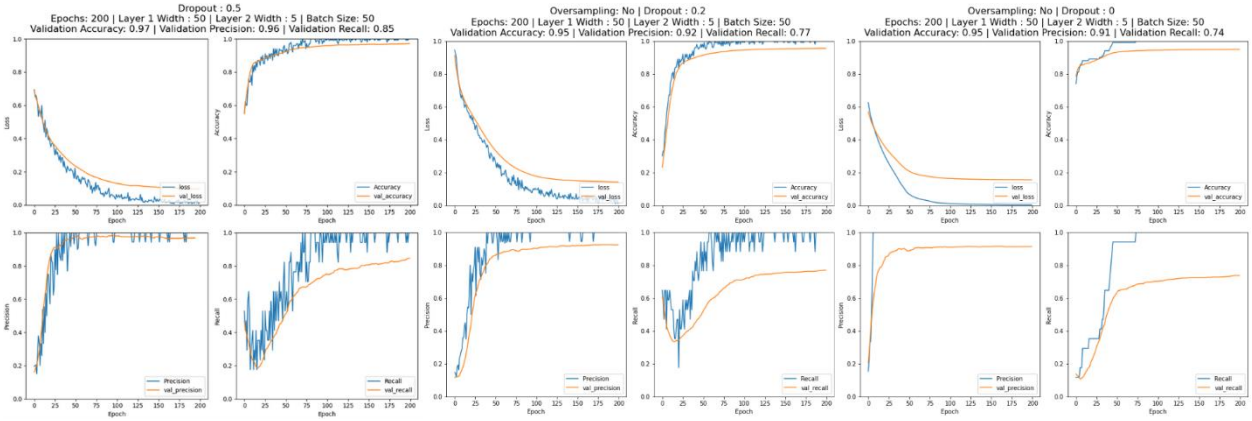

Figure A.4.3 Models tested with different dropouts (0.5, 0.2 and 0)

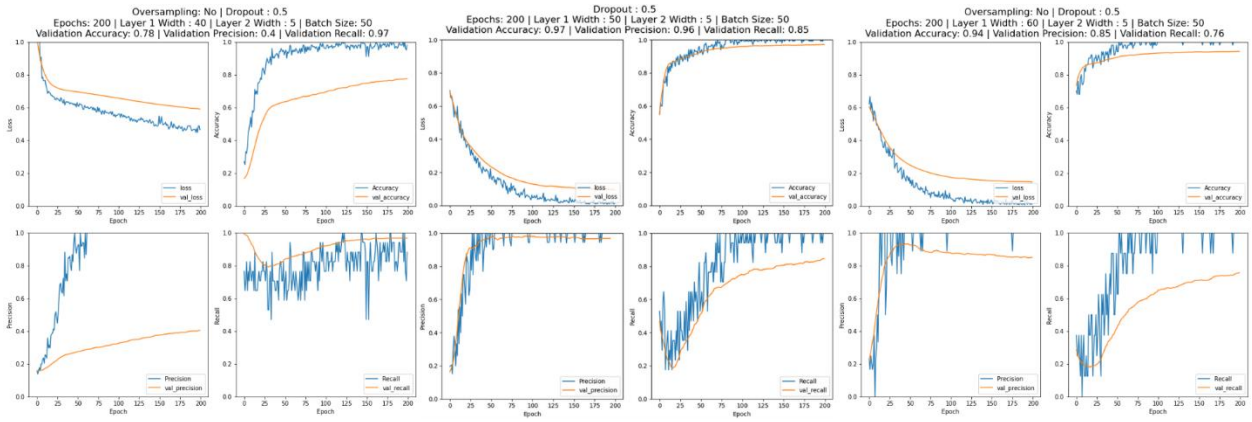

Figure A.4.4 Models tested with different layer 1 widths (40, 50 and 60)

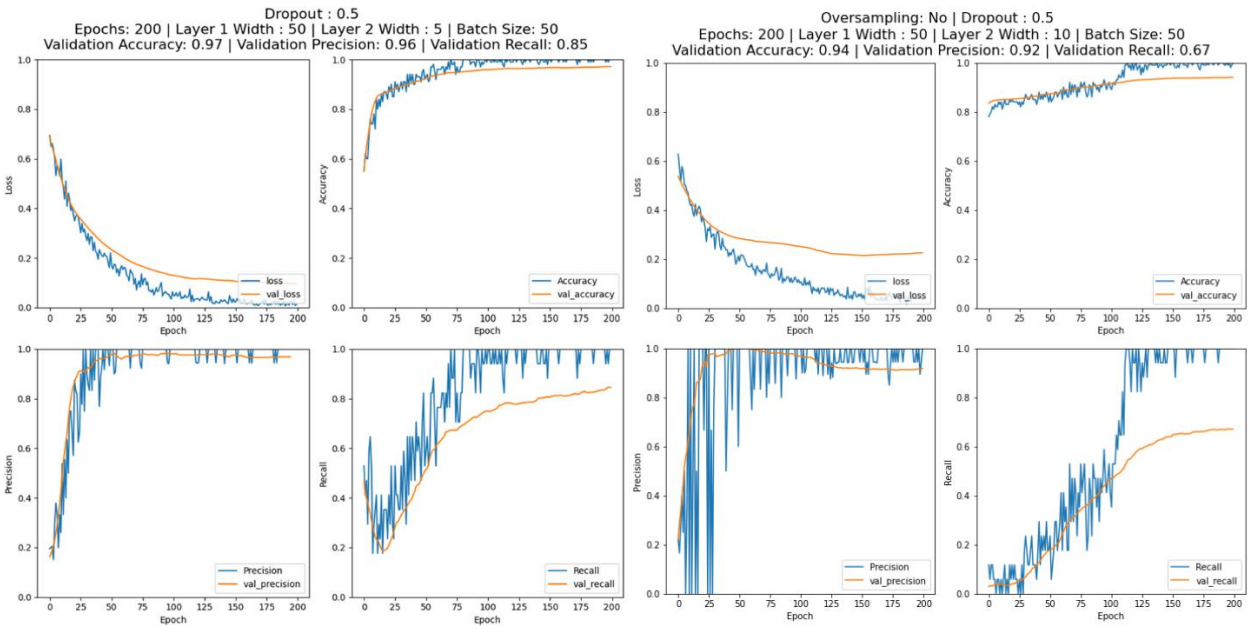

Figure A.4.5 Model tested with different layer 2 width(5 and 10)

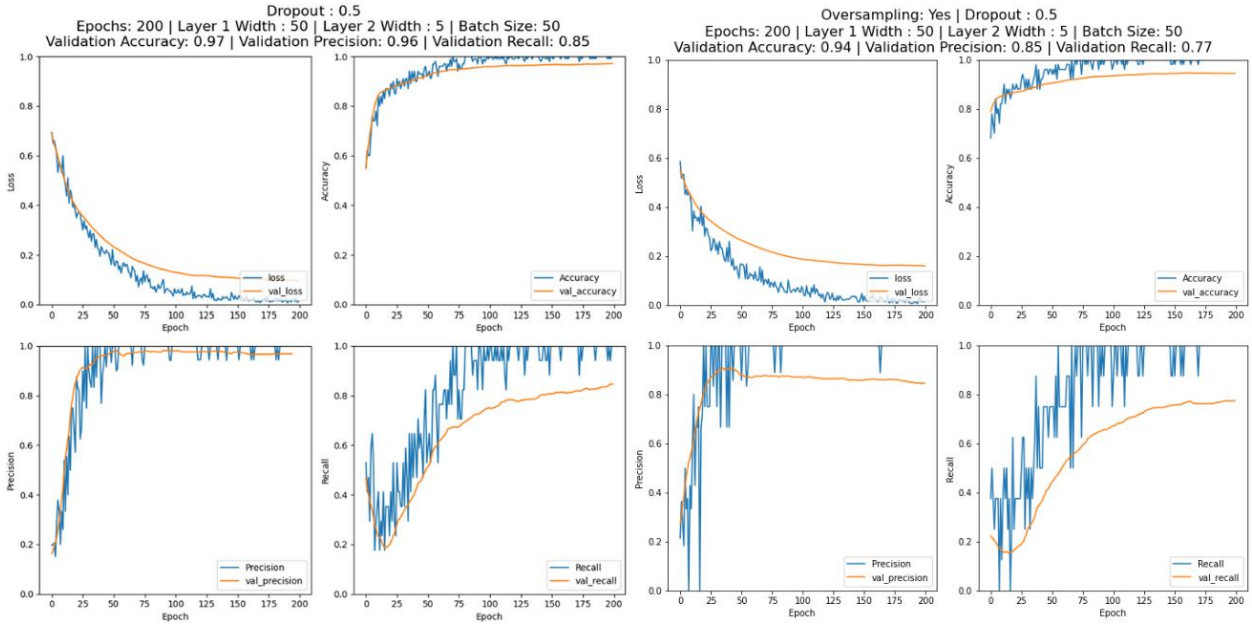

Figure A.4.6 Model performance compared with and without oversampling using SMOTE