mlFinalsCC

July 29, 2025

1 Introduction

Credit card approval is an important process to financial institutions like banks. The ability to accurately determine whether an applicant should be approved for their credit card application can significantly reduce risk of the financial insititions gaining customers that are incapable of paying back while ensure that legitimate. eligible customers receive timely credit. Usually, such decisions have been made by relying on human review of the applicants background information. However, with the appropriate data and capabilites of machine learning, it should be possible to build robust models that automate and optimise credit approval decisions.

The goal for this assignment is to build, evaluate and refine a reliable and accurate deep neural network model that can automate and optimise credit card approvals by separating approved and non-approved applications based on appropriate data such as demographic, financial. and credit history. We will be using a dataset from kaggle, while adhering to assignment restrictions such as only using dense and dropout layers.

2 Dataset

The Credit Card Approval Prediction dataset from Kaggle. It contains two csv files:

1. **application_record.csv**
   - Contains applicant demographic and financial data (e.g., income, employment status, and family status).
   - Each row is a unique applicant identified by an ID
2. **credit_record.csv**
   - Details monthly balance and repayment status for each applicant's credit history
   - Includes multiple entries per ID, reflecting different months and statuses.

By merging these files on a common identifier (ID), we derive a comprehensive dataset of demographic, financial and credit features that can be used to predict credit card approvals.

Key features

1. **Demographics**
   - gender
   - marital status
   - number of children
2. **Financial**
   - income
   - days employed
   - ownership of property
3. **Credit history**
   - monthly balance status
   - payment regularity

The dataset can be found on kaggle (https://www.kaggle.com/datasets/rikdifos/credit-card-approval-prediction).

3 Dataset Limitations

1. **Dataset has no specific "approved" column.**
   - Whether an applicant is approved or not must be inferred from credit status codes, which can be ambigious or unreliable if statuses are missing or inconsistent.
2. **Multiple records per applicant**
   - The credit record file contains multiple monthly status entries for a single applicant. This can lead to ambiguity or misleading data.
3. **Potential Class imbalance**
   - The distribution of approved vs non-approved applications may be skewed. If significantly imbalanced, the model may bias towards the majority class.

4 Constraints and Methodological Focus

1. **DLWP chapters 1 to 4**
   - As stated in the assignment, models are restricted to only dense and dropout layers in a sequential architecture.
2. **Data privacy and security**
   - In a real-world case, credit data is extremely sensitive. This project uses an anonymised dataset.
3. **Computational resources**
   - The experiments are constrained by my system's CPU, limiting the size and depth of the models tested.

5 Objectives

1. Data preprocessing
   - inspect and clean data
   - handle class imbalances (if any)
2. Model development and exploration
   - implement a baseline model
   - experiment with different types of models, e.g. shallower, deeper
3. Performance evaluation
   - Evaluate with relevant metrics (accuracy, f1 score, AUC, confusion matrix)
   - Analyse trade-offs between false approvals and missed good applicants
4. Model comparison
   - Analyse and compare different architectures or training strategies
   - Evaluate performance trade-offs (accuracy vs computational cost)
5. Model optimisation
   - Tune hyperparameters (learning rate, batch size, optimiser)
   - Train and evaluate tuned model

6 Methodology

[256]:
```
# imports
import random
import pandas as pd
import numpy as np
import tensorflow as tf
import seaborn as sns
```

```
import matplotlib.pyplot as plt
import kerastuner as kt

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.utils.class_weight import compute_class_weight
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from sklearn.metrics import classification_report, confusion_matrix, roc_curve,␣
↪auc, accuracy_score

from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
```

Fixing seed Fixing the seed ensures reproducible results, and helps make debugging easier by reducing the randomness.

[205]:
```
# fix random seed
seed = 42
np.random.seed(seed)
tf.random.set_seed(seed)
random.seed(seed)
```

Data preprocessing Credit Card Approval Dataset's 2 csv files are loaded, inspected and merged.

[206]:
```
# load the datasets
df1 = pd.read_csv('creditcardData/application_record.csv')
df2 = pd.read_csv('creditcardData/credit_record.csv')

print(df1.shape)
print(df2.shape)

print(df1.head(0))
print(df2.head(0))
```

```
(438557, 18)
(1048575, 3)
Empty DataFrame
Columns: [ID, CODE_GENDER, FLAG_OWN_CAR, FLAG_OWN_REALTY, CNT_CHILDREN,
AMT_INCOME_TOTAL, NAME_INCOME_TYPE, NAME_EDUCATION_TYPE, NAME_FAMILY_STATUS,
NAME_HOUSING_TYPE, DAYS_BIRTH, DAYS_EMPLOYED, FLAG_MOBIL, FLAG_WORK_PHONE,
FLAG_PHONE, FLAG_EMAIL, OCCUPATION_TYPE, CNT_FAM_MEMBERS]
Index: []
Empty DataFrame
Columns: [ID, MONTHS_BALANCE, STATUS]
Index: []
```

[207]:
```
# merge datasets
data_df = pd.merge(df1, df2, on='ID', how='inner')
print(data_df.head())
```

```
        ID CODE_GENDER FLAG_OWN_CAR FLAG_OWN_REALTY  CNT_CHILDREN  \
0  5008804           M            Y               Y             0
1  5008804           M            Y               Y             0
2  5008804           M            Y               Y             0
3  5008804           M            Y               Y             0
4  5008804           M            Y               Y             0

   AMT_INCOME_TOTAL NAME_INCOME_TYPE NAME_EDUCATION_TYPE NAME_FAMILY_STATUS  \
0          427500.0          Working    Higher education     Civil marriage
1          427500.0          Working    Higher education     Civil marriage
2          427500.0          Working    Higher education     Civil marriage
3          427500.0          Working    Higher education     Civil marriage
4          427500.0          Working    Higher education     Civil marriage

  NAME_HOUSING_TYPE  DAYS_BIRTH  DAYS_EMPLOYED  FLAG_MOBIL  FLAG_WORK_PHONE  \
0  Rented apartment      -12005          -4542           1                1
1  Rented apartment      -12005          -4542           1                1
2  Rented apartment      -12005          -4542           1                1
3  Rented apartment      -12005          -4542           1                1
4  Rented apartment      -12005          -4542           1                1

   FLAG_PHONE  FLAG_EMAIL OCCUPATION_TYPE  CNT_FAM_MEMBERS  MONTHS_BALANCE  \
0           0           0             NaN              2.0               0
1           0           0             NaN              2.0              -1
2           0           0             NaN              2.0              -2
3           0           0             NaN              2.0              -3
4           0           0             NaN              2.0              -4

  STATUS
0      C
1      C
2      C
3      C
4      C
```

[208]:
```
# check shape
data_df.shape
```

[208]: (777715, 20)

[209]:
```
# check for missing values
print(data_df.isnull().sum())
```

```
ID                     0
```

```
CODE_GENDER                 0
FLAG_OWN_CAR                0
FLAG_OWN_REALTY             0
CNT_CHILDREN                0
AMT_INCOME_TOTAL            0
NAME_INCOME_TYPE            0
NAME_EDUCATION_TYPE         0
NAME_FAMILY_STATUS          0
NAME_HOUSING_TYPE           0
DAYS_BIRTH                  0
DAYS_EMPLOYED               0
FLAG_MOBIL                  0
FLAG_WORK_PHONE             0
FLAG_PHONE                  0
FLAG_EMAIL                  0
OCCUPATION_TYPE        240048
CNT_FAM_MEMBERS             0
MONTHS_BALANCE              0
STATUS                      0
dtype: int64
```

[210]:
```
# check for duplicate rows and drop
data_df = data_df.drop_duplicates()
data_df.shape
```

[210]: (777715, 20)

[211]:
```
# check for missing values again
print(data_df.isnull().sum())
```

```
ID                          0
CODE_GENDER                 0
FLAG_OWN_CAR                0
FLAG_OWN_REALTY             0
CNT_CHILDREN                0
AMT_INCOME_TOTAL            0
NAME_INCOME_TYPE            0
NAME_EDUCATION_TYPE         0
NAME_FAMILY_STATUS          0
NAME_HOUSING_TYPE           0
DAYS_BIRTH                  0
DAYS_EMPLOYED               0
FLAG_MOBIL                  0
FLAG_WORK_PHONE             0
FLAG_PHONE                  0
FLAG_EMAIL                  0
OCCUPATION_TYPE        240048
CNT_FAM_MEMBERS             0
```

```
MONTHS_BALANCE              0
STATUS                      0
dtype: int64
```

[212]:
```
# check missing occupation rows ID and fill with forward fill
data_df['OCCUPATION_TYPE'] = data_df.groupby('ID')['OCCUPATION_TYPE'].
 ↪transform(lambda x: x.ffill().bfill())

data_df.isnull().sum()
```

```
C:\Users\Admin\AppData\Local\Temp\ipykernel_5756\4124487580.py:2: FutureWarning:
Downcasting object dtype arrays on .fillna, .ffill, .bfill is deprecated and
will change in a future version. Call result.infer_objects(copy=False) instead.
To opt-in to the future behavior, set
`pd.set_option('future.no_silent_downcasting', True)`
  data_df['OCCUPATION_TYPE'] =
data_df.groupby('ID')['OCCUPATION_TYPE'].transform(lambda x: x.ffill().bfill())
```

[212]:
```
ID                          0
CODE_GENDER                 0
FLAG_OWN_CAR                0
FLAG_OWN_REALTY             0
CNT_CHILDREN                0
AMT_INCOME_TOTAL            0
NAME_INCOME_TYPE            0
NAME_EDUCATION_TYPE         0
NAME_FAMILY_STATUS          0
NAME_HOUSING_TYPE           0
DAYS_BIRTH                  0
DAYS_EMPLOYED               0
FLAG_MOBIL                  0
FLAG_WORK_PHONE             0
FLAG_PHONE                  0
FLAG_EMAIL                  0
OCCUPATION_TYPE        240048
CNT_FAM_MEMBERS             0
MONTHS_BALANCE              0
STATUS                      0
dtype: int64
```

[213]:
```
# drop rest of missing values
data_df = data_df.dropna()

data_df.isnull().sum()
```

[213]:
```
ID                  0
CODE_GENDER         0
FLAG_OWN_CAR        0
```

```
FLAG_OWN_REALTY         0
CNT_CHILDREN            0
AMT_INCOME_TOTAL        0
NAME_INCOME_TYPE        0
NAME_EDUCATION_TYPE     0
NAME_FAMILY_STATUS      0
NAME_HOUSING_TYPE       0
DAYS_BIRTH              0
DAYS_EMPLOYED           0
FLAG_MOBIL              0
FLAG_WORK_PHONE         0
FLAG_PHONE              0
FLAG_EMAIL              0
OCCUPATION_TYPE         0
CNT_FAM_MEMBERS         0
MONTHS_BALANCE          0
STATUS                  0
dtype: int64
```

As there are no more duplicates or null values in any row or column, we can move on to converting the categorical nominal columns into one-hot encoded variables.

This is so that the model is able to properly interpret and learn from these features.

[214]:
```
# categorical features for one-hot encoding
categorical_cols = [
    'CODE_GENDER', 'FLAG_OWN_CAR', 'FLAG_OWN_REALTY',
    'NAME_INCOME_TYPE', 'NAME_EDUCATION_TYPE', 'NAME_FAMILY_STATUS',
    'NAME_HOUSING_TYPE', 'OCCUPATION_TYPE'
]

# dummify
data_df = pd.get_dummies(data_df, columns=categorical_cols, drop_first=True)

data_df.shape
```

[214]:
```
(537667, 49)
```

Now that the nominal columns have been converted, we work on the ordinal columns. First we create an "Approved" column to use as target variable.

[215]:
```
# create the target variable and label 1 as approved where STATUS is C or X
data_df['Approved'] = np.where(((data_df['STATUS'] == 'C') == 1) |␣
 ↪((data_df['STATUS'] == 'X') == 1), 1, 0)

data_df['Approved'].value_counts()
```

[215]:
```
Approved
1    328352
```

```
0    209315
Name: count, dtype: int64
```

Then we map and aggregate the rest of the status values so that it still makes sense to the ml model.

[216]:
```
# map and aggregate status values to numeric scores
status_mapping = {
    '0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5,  # overdue severity
    'C': -1,  # paid off that month
    'X': -2   # no loan that month
}

df2['STATUS_NUM'] = df2['STATUS'].map(status_mapping)

# aggregate per applicant (ID)
agg_status = df2.groupby('ID').agg({
    'STATUS_NUM': ['max', 'mean', lambda x: (x > 0).sum(), lambda x: (x == -1).
 ↪sum(), lambda x: (x == -2).sum()]
})

# rename columns
agg_status.columns = ['STATUS_MAX', 'STATUS_MEAN', 'STATUS_COUNT_OVERDUE',␣
 ↪'STATUS_COUNT_C', 'STATUS_COUNT_X']

# merge back into the dataset
data_df = data_df.merge(agg_status, on='ID', how='left')

# drop original status column
data_df.drop(columns=['STATUS'], inplace=True)
```

[217]:
```
# check currect columns
print(data_df.columns)
print(data_df.shape)
```

```
Index(['ID', 'CNT_CHILDREN', 'AMT_INCOME_TOTAL', 'DAYS_BIRTH', 'DAYS_EMPLOYED',
       'FLAG_MOBIL', 'FLAG_WORK_PHONE', 'FLAG_PHONE', 'FLAG_EMAIL',
       'CNT_FAM_MEMBERS', 'MONTHS_BALANCE', 'CODE_GENDER_M', 'FLAG_OWN_CAR_Y',
       'FLAG_OWN_REALTY_Y', 'NAME_INCOME_TYPE_Pensioner',
       'NAME_INCOME_TYPE_State servant', 'NAME_INCOME_TYPE_Student',
       'NAME_INCOME_TYPE_Working', 'NAME_EDUCATION_TYPE_Higher education',
       'NAME_EDUCATION_TYPE_Incomplete higher',
       'NAME_EDUCATION_TYPE_Lower secondary',
       'NAME_EDUCATION_TYPE_Secondary / secondary special',
       'NAME_FAMILY_STATUS_Married', 'NAME_FAMILY_STATUS_Separated',
       'NAME_FAMILY_STATUS_Single / not married', 'NAME_FAMILY_STATUS_Widow',
       'NAME_HOUSING_TYPE_House / apartment',
       'NAME_HOUSING_TYPE_Municipal apartment',
```

```
       'NAME_HOUSING_TYPE_Office apartment',
       'NAME_HOUSING_TYPE_Rented apartment', 'NAME_HOUSING_TYPE_With parents',
       'OCCUPATION_TYPE_Cleaning staff', 'OCCUPATION_TYPE_Cooking staff',
       'OCCUPATION_TYPE_Core staff', 'OCCUPATION_TYPE_Drivers',
       'OCCUPATION_TYPE_HR staff', 'OCCUPATION_TYPE_High skill tech staff',
       'OCCUPATION_TYPE_IT staff', 'OCCUPATION_TYPE_Laborers',
       'OCCUPATION_TYPE_Low-skill Laborers', 'OCCUPATION_TYPE_Managers',
       'OCCUPATION_TYPE_Medicine staff',
       'OCCUPATION_TYPE_Private service staff',
       'OCCUPATION_TYPE_Realty agents', 'OCCUPATION_TYPE_Sales staff',
       'OCCUPATION_TYPE_Secretaries', 'OCCUPATION_TYPE_Security staff',
       'OCCUPATION_TYPE_Waiters/barmen staff', 'Approved', 'STATUS_MAX',
       'STATUS_MEAN', 'STATUS_COUNT_OVERDUE', 'STATUS_COUNT_C',
       'STATUS_COUNT_X'],
      dtype='object')
(537667, 54)
```

Now that we have handled the nominal and ordinal columns, we scale the numerical features.

[218]:
```
# identify numerical columns to scale
num_cols = data_df.select_dtypes(include=['int64', 'float64']).columns.
 ↪difference(['ID', 'Approved'])

scaler = StandardScaler()
data_df[num_cols] = scaler.fit_transform(data_df[num_cols])

# check that num of cols is still the same
print(data_df.shape)
```

```
(537667, 54)
```

We convert boolean columns to integers as the models expect numeric inputs, and not T/F.

[220]:
```
# convert boolean columns to integers
bool_cols = data_df.select_dtypes(include=['bool']).columns
data_df[bool_cols] = data_df[bool_cols].astype(int)
```

[221]:
```
# last check
data_df.head(1)
```

[221]:

	ID	CNT_CHILDREN	AMT_INCOME_TOTAL	DAYS_BIRTH	DAYS_EMPLOYED	\
0	5008806	-0.643601	-0.812541	-1.891761	0.680069	

	FLAG_MOBIL	FLAG_WORK_PHONE	FLAG_PHONE	FLAG_EMAIL	CNT_FAM_MEMBERS	…	\
0	0.0	-0.626108	-0.652929	-0.334683	-0.323497	…	

	OCCUPATION_TYPE_Sales staff	OCCUPATION_TYPE_Secretaries	\
0	0	0	

```
   OCCUPATION_TYPE_Security staff  OCCUPATION_TYPE_Waiters/barmen staff  \
0                               1                                     0

   Approved  STATUS_MAX  STATUS_MEAN  STATUS_COUNT_OVERDUE  STATUS_COUNT_C  \
0         1     0.00692    -0.892829             -0.222849       -0.527939

   STATUS_COUNT_X
0        0.881483

[1 rows x 54 columns]
```

Now that all the data has been preprocessed, we move onto splitting it into the train/test sets.

[222]:
```
# define features and target
X = data_df.drop(columns=['ID', 'Approved'])
y = data_df['Approved']

# split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

print("X_train shape:", X_train.shape)
print("y_train distribution:\n", y_train.value_counts())
```

```
X_train shape: (430133, 52)
y_train distribution:
 Approved
1    262731
0    167402
Name: count, dtype: int64
```

From the above distribution numbers, there is a mild class imbalance of about 60% to 40%. Although not severe, we will still implement class weighting to address this issue.

[225]:
```
# compute class weights
class_weights = compute_class_weight(
    class_weight="balanced",
    classes=np.unique(y_train),
    y=y_train
)

# convert to dictionary format
class_weights_dict = {i: class_weights[i] for i in range(len(class_weights))}
class_weights_dict
```

[225]:
```
{0: 1.284730767852236, 1: 0.8185806014516749}
```

Now that we have addressed the class imbalance, we go on to build our baseline model.

[ ]:
```
# baseline model, 2 hidden and 2 dropoutf layers
baseline_model = Sequential([
    Dense(32, activation='relu', input_shape=(X_train.shape[1],)), # input layer
    Dropout(0.5),
    Dense(16, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])

baseline_model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
    loss='binary_crossentropy',
    metrics=['accuracy'])

baseline_model.summary()
```

```
c:\Users\Admin\anaconda3\Lib\site-packages\keras\src\layers\core\dense.py:87:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
dense_2 (Dense)	(None, 32)	1,696
dropout_1 (Dropout)	(None, 32)	0
dense_3 (Dense)	(None, 16)	528
dropout_2 (Dropout)	(None, 16)	0
dense_4 (Dense)	(None, 1)	17

Total params: 2,241 (8.75 KB)

Trainable params: 2,241 (8.75 KB)

Non-trainable params: 0 (0.00 B)

[228]:
```
# train model with class weights
baseline_hist = baseline_model.fit(
    X_train, y_train,
    epochs=20,
    batch_size=32,
    validation_split=0.2,
    class_weight=class_weights_dict,
    verbose=1
)
```

```
Epoch 1/20
10754/10754             13s 1ms/step
- accuracy: 0.6547 - loss: 0.6054 - val_accuracy: 0.8476 - val_loss: 0.3336
Epoch 2/20
10754/10754             12s 1ms/step
- accuracy: 0.8310 - loss: 0.3744 - val_accuracy: 0.8639 - val_loss: 0.2887
Epoch 3/20
10754/10754             12s 1ms/step
- accuracy: 0.8497 - loss: 0.3181 - val_accuracy: 0.8746 - val_loss: 0.2642
Epoch 4/20
10754/10754             11s 1ms/step
- accuracy: 0.8617 - loss: 0.2873 - val_accuracy: 0.8815 - val_loss: 0.2490
Epoch 5/20
10754/10754             11s 1ms/step
- accuracy: 0.8692 - loss: 0.2703 - val_accuracy: 0.8852 - val_loss: 0.2406
Epoch 6/20
10754/10754             12s 1ms/step
- accuracy: 0.8734 - loss: 0.2589 - val_accuracy: 0.8867 - val_loss: 0.2360
Epoch 7/20
10754/10754             11s 1ms/step
- accuracy: 0.8756 - loss: 0.2528 - val_accuracy: 0.8874 - val_loss: 0.2327
Epoch 8/20
10754/10754             11s 1ms/step
- accuracy: 0.8773 - loss: 0.2475 - val_accuracy: 0.8870 - val_loss: 0.2297
Epoch 9/20
10754/10754             11s 1ms/step
- accuracy: 0.8798 - loss: 0.2432 - val_accuracy: 0.8871 - val_loss: 0.2274
Epoch 10/20
10754/10754             11s 1ms/step
- accuracy: 0.8792 - loss: 0.2412 - val_accuracy: 0.8865 - val_loss: 0.2269
Epoch 11/20
10754/10754             11s 1ms/step
- accuracy: 0.8797 - loss: 0.2393 - val_accuracy: 0.8869 - val_loss: 0.2255
Epoch 12/20
10754/10754             12s 1ms/step
- accuracy: 0.8808 - loss: 0.2362 - val_accuracy: 0.8867 - val_loss: 0.2253
Epoch 13/20
10754/10754             11s 1ms/step
```

```
- accuracy: 0.8808 - loss: 0.2351 - val_accuracy: 0.8868 - val_loss: 0.2246
Epoch 14/20
10754/10754             12s 1ms/step
- accuracy: 0.8802 - loss: 0.2350 - val_accuracy: 0.8881 - val_loss: 0.2234
Epoch 15/20
10754/10754             12s 1ms/step
- accuracy: 0.8812 - loss: 0.2343 - val_accuracy: 0.8873 - val_loss: 0.2235
Epoch 16/20
10754/10754             13s 1ms/step
- accuracy: 0.8819 - loss: 0.2324 - val_accuracy: 0.8870 - val_loss: 0.2240
Epoch 17/20
10754/10754             13s 1ms/step
- accuracy: 0.8813 - loss: 0.2329 - val_accuracy: 0.8878 - val_loss: 0.2223
Epoch 18/20
10754/10754             13s 1ms/step
- accuracy: 0.8812 - loss: 0.2322 - val_accuracy: 0.8873 - val_loss: 0.2232
Epoch 19/20
10754/10754             14s 1ms/step
- accuracy: 0.8821 - loss: 0.2325 - val_accuracy: 0.8876 - val_loss: 0.2225
Epoch 20/20
10754/10754             11s 1ms/step
- accuracy: 0.8809 - loss: 0.2320 - val_accuracy: 0.8877 - val_loss: 0.2225
```

After building and training our baseline model, we evaluate and plot the model on test data.

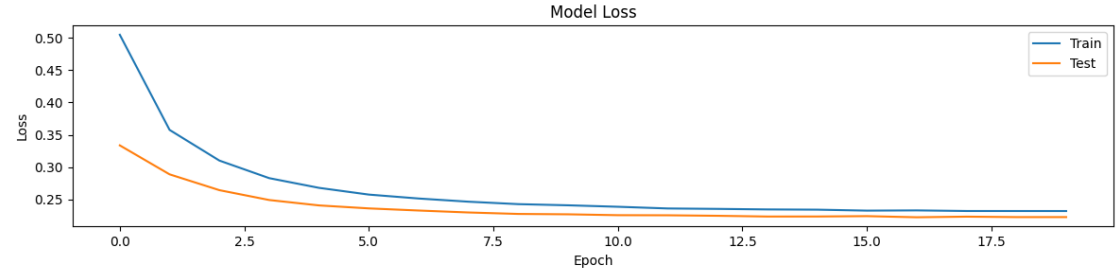
[ ]:
```python
# function to print metrics and graphs for model
def metrics_and_graphs(model, history):
    # predictions
    y_scores = model.predict(X_test)
    y_pred = (y_scores > 0.5).astype("int32")

    # classification report
    print(classification_report(y_test, y_pred))

    # evaluation of loss and acc
    loss, accuracy = model.evaluate(X_test, y_test, verbose=0)
    print("Test Loss:", loss)
    print("Test Accuracy:", accuracy)

    # plot loss and acc graphs
    plt.figure(figsize=(12, 6))
    # 1. loss
    plt.subplot(2, 1, 1)
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title('Model Loss')
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
```

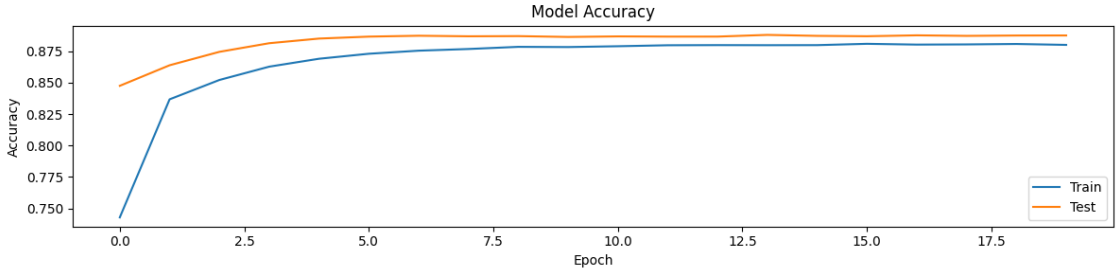
```
    plt.legend(['Train', 'Test'])

    # 2. acc
    plt.subplot(2, 1, 2)
    plt.plot(history.history['accuracy'])
    plt.plot(history.history['val_accuracy'])
    plt.title('Model Accuracy')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend(['Train', 'Test'])

    plt.tight_layout()
    plt.show()

    # confusion matrix & ROC curve side by side
    fig, ax = plt.subplots(1, 2, figsize=(12, 5))  # 1 row, 2 columns

    # 3. confusion matrix
    cm = confusion_matrix(y_test, y_pred)
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
                xticklabels=["Not Approved", "Approved"],
                yticklabels=["Not Approved", "Approved"], ax=ax[0])
    ax[0].set_xlabel("Predicted")
    ax[0].set_ylabel("Actual")
    ax[0].set_title("Confusion Matrix")

    # 4. ROC curve
    if len(np.unique(y_test)) > 1:  # check if both classes exist
        fpr, tpr, thresholds = roc_curve(y_test, y_scores)
        roc_auc = auc(fpr, tpr)
        ax[1].plot(fpr, tpr, label=f"ROC (AUC={roc_auc:.2f})", linewidth=2)
        ax[1].plot([0,1], [0,1], 'r--')  # Diagonal reference line
        ax[1].set_xlabel("False Positive Rate")
        ax[1].set_ylabel("True Positive Rate")
        ax[1].set_title("ROC Curve")
        ax[1].legend()
    else:
        ax[1].set_visible(False)  # hide if only one class exists
        print("Skipping ROC curve: Only one class present in test set.")

    plt.tight_layout()
    plt.show()
```

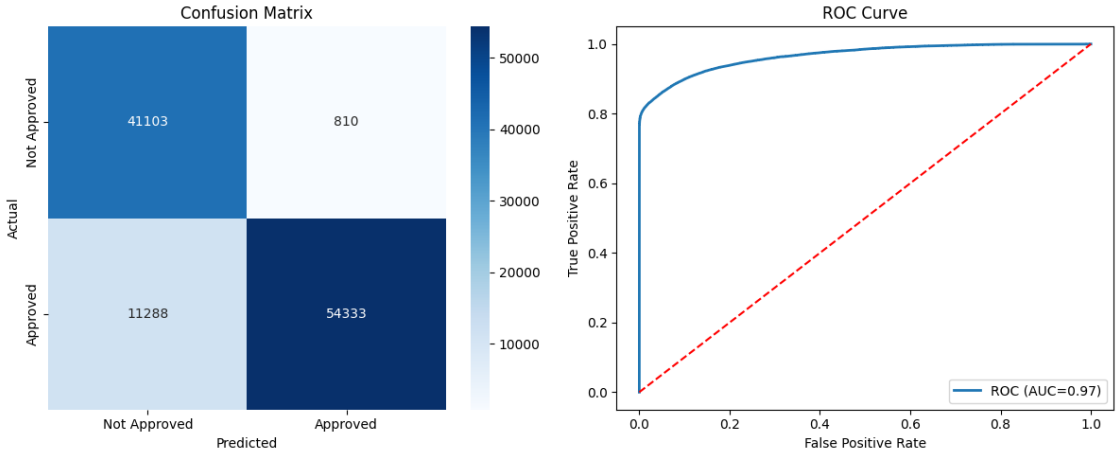
[244]:
```
# print metrics and graphs for baseline model
metrics_and_graphs(baseline_model, baseline_hist)
```

```
3361/3361             2s 507us/step
              precision    recall  f1-score   support
```

```
              precision    recall  f1-score   support

           0       0.78      0.98      0.87     41913
           1       0.99      0.83      0.90     65621

    accuracy                           0.89    107534
   macro avg       0.88      0.90      0.89    107534
weighted avg       0.91      0.89      0.89    107534

Test Loss: 0.2237771600484848
Test Accuracy: 0.8874960541725159
```







Now that we have evaluated our baseline model, I want to compare it against traditional ML models in order to find out if deep learning is worth the training time and complexity.

I will only be testing it against 2 models for simplicities sake, namely the logistic regression and random forest model.

[ ]:
```
# logreg
lr_model = LogisticRegression(max_iter=1000)
lr_model.fit(X_train, y_train)
y_pred_lr = lr_model.predict(X_test)
lr_acc = accuracy_score(y_test, y_pred_lr)

# random forest
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)
y_pred_rf = rf_model.predict(X_test)
rf_acc = accuracy_score(y_test, y_pred_rf)

# print accuracy against the baseline model
print("Logistic Regression Accuracy:", lr_acc)
print("Random Forest Accuracy:", rf_acc)
```

```
Logistic Regression Accuracy: 0.8338943961909722
Random Forest Accuracy: 0.9155615898227537
Baseline Accuracy: 0.89
```

Our baseline mode has an accuracy of 0.8874, and as we can see from the comparison against the 2 traditional models, it has come in second place.

We will now try different deep learning architectures to find out which architecture could potentially perform better than the Random Forest's accuracy score of 0.9155, or could perform the best out of the deep learning models.

I will be building a deeper and a wider model to compare against the previous models. I have decided not to include a separate shallow model as the baseline model basically acts as one with only 2 hidden layers

[248]:
```
# create deeper model (same dropout rate of 0.5 as the baseline model)
deeper_model = Sequential([
    Dense(128, activation='relu', input_shape=(X_train.shape[1],)),
    Dropout(0.5),
    Dense(64, activation='relu'),
    Dropout(0.5),
    Dense(32, activation='relu'),
    Dense(1, activation='sigmoid')
])

deeper_model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
    loss='binary_crossentropy',
    metrics=['accuracy'])
```

```
deeper_model.summary()
```

```
c:\Users\Admin\anaconda3\Lib\site-packages\keras\src\layers\core\dense.py:87:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential_4"

Layer (type)	Output Shape	Param #
dense_12 (Dense)	(None, 128)	6,784
dropout_6 (Dropout)	(None, 128)	0
dense_13 (Dense)	(None, 64)	8,256
dropout_7 (Dropout)	(None, 64)	0
dense_14 (Dense)	(None, 32)	2,080
dense_15 (Dense)	(None, 1)	33

Total params: 17,153 (67.00 KB)

Trainable params: 17,153 (67.00 KB)

Non-trainable params: 0 (0.00 B)

[249]:
```
# train deeper model
deeper_hist = deeper_model.fit(
    X_train, y_train,
    epochs=20,
    batch_size=32,
    validation_split=0.2,
    class_weight=class_weights_dict,
    verbose=1
)
```

```
Epoch 1/20
10754/10754             15s 1ms/step
- accuracy: 0.7586 - loss: 0.4710 - val_accuracy: 0.8634 - val_loss: 0.2910
Epoch 2/20
```

```
10754/10754            13s 1ms/step
- accuracy: 0.8560 - loss: 0.3008 - val_accuracy: 0.8777 - val_loss: 0.2538
Epoch 3/20
10754/10754            13s 1ms/step
- accuracy: 0.8705 - loss: 0.2630 - val_accuracy: 0.8854 - val_loss: 0.2389
Epoch 4/20
10754/10754            14s 1ms/step
- accuracy: 0.8791 - loss: 0.2441 - val_accuracy: 0.8892 - val_loss: 0.2301
Epoch 5/20
10754/10754            13s 1ms/step
- accuracy: 0.8849 - loss: 0.2332 - val_accuracy: 0.8912 - val_loss: 0.2251
Epoch 6/20
10754/10754            13s 1ms/step
- accuracy: 0.8887 - loss: 0.2256 - val_accuracy: 0.8930 - val_loss: 0.2228
Epoch 7/20
10754/10754            13s 1ms/step
- accuracy: 0.8920 - loss: 0.2221 - val_accuracy: 0.8961 - val_loss: 0.2204
Epoch 8/20
10754/10754            17s 2ms/step
- accuracy: 0.8942 - loss: 0.2189 - val_accuracy: 0.8982 - val_loss: 0.2189
Epoch 9/20
10754/10754            14s 1ms/step
- accuracy: 0.8963 - loss: 0.2164 - val_accuracy: 0.8999 - val_loss: 0.2176
Epoch 10/20
10754/10754            14s 1ms/step
- accuracy: 0.8971 - loss: 0.2151 - val_accuracy: 0.9002 - val_loss: 0.2167
Epoch 11/20
10754/10754            14s 1ms/step
- accuracy: 0.8983 - loss: 0.2135 - val_accuracy: 0.9008 - val_loss: 0.2159
Epoch 12/20
10754/10754            14s 1ms/step
- accuracy: 0.8984 - loss: 0.2128 - val_accuracy: 0.9009 - val_loss: 0.2145
Epoch 13/20
10754/10754            13s 1ms/step
- accuracy: 0.8989 - loss: 0.2114 - val_accuracy: 0.9014 - val_loss: 0.2138
Epoch 14/20
10754/10754            13s 1ms/step
- accuracy: 0.8992 - loss: 0.2103 - val_accuracy: 0.9018 - val_loss: 0.2136
Epoch 15/20
10754/10754            12s 1ms/step
- accuracy: 0.8996 - loss: 0.2103 - val_accuracy: 0.9012 - val_loss: 0.2137
Epoch 16/20
10754/10754            12s 1ms/step
- accuracy: 0.8999 - loss: 0.2096 - val_accuracy: 0.9023 - val_loss: 0.2122
Epoch 17/20
10754/10754            12s 1ms/step
- accuracy: 0.9004 - loss: 0.2089 - val_accuracy: 0.9021 - val_loss: 0.2118
Epoch 18/20
```

```
10754/10754              12s 1ms/step
- accuracy: 0.8998 - loss: 0.2093 - val_accuracy: 0.9023 - val_loss: 0.2112
Epoch 19/20
10754/10754              12s 1ms/step
- accuracy: 0.9004 - loss: 0.2086 - val_accuracy: 0.9027 - val_loss: 0.2115
Epoch 20/20
10754/10754              12s 1ms/step
- accuracy: 0.9006 - loss: 0.2073 - val_accuracy: 0.9031 - val_loss: 0.2106
```

[250]:
```
# create wider model (same dropout rate of 0.5 as the baseline model)
wider_model = Sequential([
    Dense(128, activation='relu', input_shape=(X_train.shape[1],)),
    Dropout(0.5),
    Dense(128, activation='relu'),
    Dense(1, activation='sigmoid')
])

wider_model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
    loss='binary_crossentropy',
    metrics=['accuracy'])

wider_model.summary()
```

```
c:\Users\Admin\anaconda3\Lib\site-packages\keras\src\layers\core\dense.py:87:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential_5"

Layer (type)	Output Shape	Param #
dense_16 (Dense)	(None, 128)	6,784
dropout_8 (Dropout)	(None, 128)	0
dense_17 (Dense)	(None, 128)	16,512
dense_18 (Dense)	(None, 1)	129

Total params: 23,425 (91.50 KB)

Trainable params: 23,425 (91.50 KB)

Non-trainable params: 0 (0.00 B)

[251]:
```
# train wider model
wider_hist = wider_model.fit(
    X_train, y_train,
    epochs=20,
    batch_size=32,
    validation_split=0.2,
    class_weight=class_weights_dict,
    verbose=1
)
```

```
Epoch 1/20
10754/10754             12s 1ms/step
- accuracy: 0.8061 - loss: 0.4074 - val_accuracy: 0.8687 - val_loss: 0.2793
Epoch 2/20
10754/10754             12s 1ms/step
- accuracy: 0.8626 - loss: 0.2836 - val_accuracy: 0.8834 - val_loss: 0.2453
Epoch 3/20
10754/10754             12s 1ms/step
- accuracy: 0.8765 - loss: 0.2514 - val_accuracy: 0.8906 - val_loss: 0.2326
Epoch 4/20
10754/10754             12s 1ms/step
- accuracy: 0.8838 - loss: 0.2353 - val_accuracy: 0.8934 - val_loss: 0.2261
Epoch 5/20
10754/10754             12s 1ms/step
- accuracy: 0.8893 - loss: 0.2257 - val_accuracy: 0.8952 - val_loss: 0.2227
Epoch 6/20
10754/10754             11s 1ms/step
- accuracy: 0.8927 - loss: 0.2192 - val_accuracy: 0.8973 - val_loss: 0.2200
Epoch 7/20
10754/10754             12s 1ms/step
- accuracy: 0.8954 - loss: 0.2157 - val_accuracy: 0.8977 - val_loss: 0.2182
Epoch 8/20
10754/10754             12s 1ms/step
- accuracy: 0.8971 - loss: 0.2130 - val_accuracy: 0.8990 - val_loss: 0.2177
Epoch 9/20
10754/10754             12s 1ms/step
- accuracy: 0.8984 - loss: 0.2109 - val_accuracy: 0.8993 - val_loss: 0.2156
Epoch 10/20
10754/10754             12s 1ms/step
- accuracy: 0.8990 - loss: 0.2100 - val_accuracy: 0.8995 - val_loss: 0.2153
Epoch 11/20
10754/10754             21s 1ms/step
- accuracy: 0.8996 - loss: 0.2089 - val_accuracy: 0.8998 - val_loss: 0.2145
Epoch 12/20
```

```
10754/10754              12s 1ms/step
- accuracy: 0.9002 - loss: 0.2079 - val_accuracy: 0.9002 - val_loss: 0.2136
Epoch 13/20
10754/10754              12s 1ms/step
- accuracy: 0.9004 - loss: 0.2070 - val_accuracy: 0.9001 - val_loss: 0.2127
Epoch 14/20
10754/10754              12s 1ms/step
- accuracy: 0.9009 - loss: 0.2060 - val_accuracy: 0.9004 - val_loss: 0.2132
Epoch 15/20
10754/10754              12s 1ms/step
- accuracy: 0.9010 - loss: 0.2058 - val_accuracy: 0.9009 - val_loss: 0.2120
Epoch 16/20
10754/10754              12s 1ms/step
- accuracy: 0.9014 - loss: 0.2052 - val_accuracy: 0.9001 - val_loss: 0.2124
Epoch 17/20
10754/10754              12s 1ms/step
- accuracy: 0.9015 - loss: 0.2046 - val_accuracy: 0.9005 - val_loss: 0.2121
Epoch 18/20
10754/10754              12s 1ms/step
- accuracy: 0.9019 - loss: 0.2041 - val_accuracy: 0.9006 - val_loss: 0.2113
Epoch 19/20
10754/10754              12s 1ms/step
- accuracy: 0.9014 - loss: 0.2041 - val_accuracy: 0.9011 - val_loss: 0.2119
Epoch 20/20
10754/10754              12s 1ms/step
- accuracy: 0.9025 - loss: 0.2033 - val_accuracy: 0.9008 - val_loss: 0.2117
```

Now that both deeper and wider models have been trained, we plot the metrics for comparison.

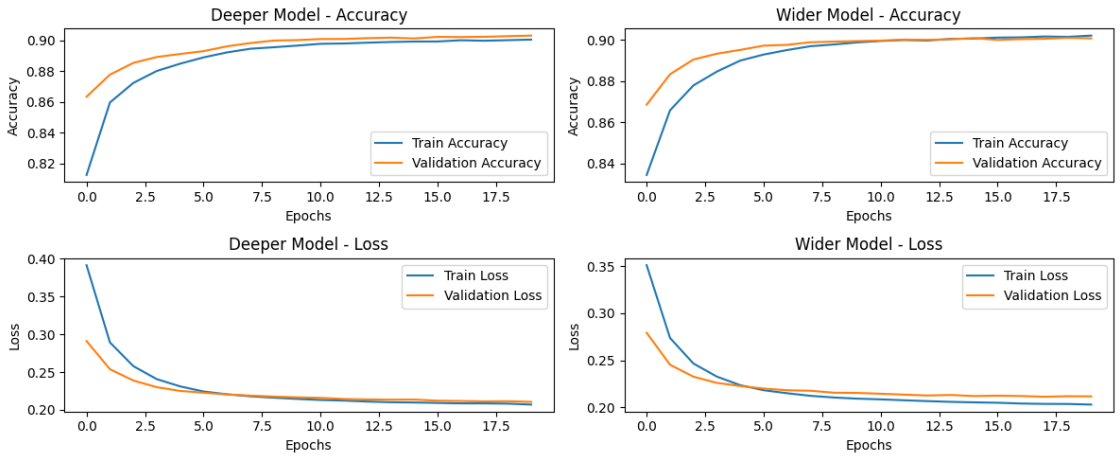
[254]:
```python
# function to compare trained models
def compare_trained_models(models, model_names, histories, X_test, y_test):
    results = {}

    plt.figure(figsize=(12, 5))

    # compare acc & loss side by side
    for i, (model, name, history) in enumerate(zip(models, model_names,␣
 ↪histories)):
        # Get predictions
        y_scores = model.predict(X_test)
        y_pred = (y_scores > 0.5).astype("int32")

        # compute accuracy
        test_acc = accuracy_score(y_test, y_pred)
        results[name] = test_acc

        # accuracy Curve
        plt.subplot(2, len(models), i + 1)
```

```
        plt.plot(history.history['accuracy'], label='Train Accuracy')
        plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
        plt.title(f'{name} - Accuracy')
        plt.xlabel('Epochs')
        plt.ylabel('Accuracy')
        plt.legend()

        # loss Curve
        plt.subplot(2, len(models), i + 1 + len(models))
        plt.plot(history.history['loss'], label='Train Loss')
        plt.plot(history.history['val_loss'], label='Validation Loss')
        plt.title(f'{name} - Loss')
        plt.xlabel('Epochs')
        plt.ylabel('Loss')
        plt.legend()

    plt.tight_layout()
    plt.show()

    # compare confusion matrix & roc curve side by side
    fig, ax = plt.subplots(len(models), 2, figsize=(12, 6 * len(models)))

    for i, (model, name) in enumerate(zip(models, model_names)):
        # get predictions
        y_scores = model.predict(X_test)
        y_pred = (y_scores > 0.5).astype("int32")

        # confusion matrix
        cm = confusion_matrix(y_test, y_pred)
        sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
                    xticklabels=["Not Approved", "Approved"],
                    yticklabels=["Not Approved", "Approved"], ax=ax[i, 0])
        ax[i, 0].set_xlabel("Predicted")
        ax[i, 0].set_ylabel("Actual")
        ax[i, 0].set_title(f"Confusion Matrix - {name}")

        # ROC curve
        if len(set(y_test)) > 1:
            fpr, tpr, _ = roc_curve(y_test, y_scores)
            roc_auc = auc(fpr, tpr)
            ax[i, 1].plot(fpr, tpr, label=f"AUC={roc_auc:.2f}")
            ax[i, 1].plot([0,1], [0,1], 'r--')
            ax[i, 1].set_xlabel("False Positive Rate")
            ax[i, 1].set_ylabel("True Positive Rate")
            ax[i, 1].set_title(f"ROC Curve - {name}")
            ax[i, 1].legend()
```

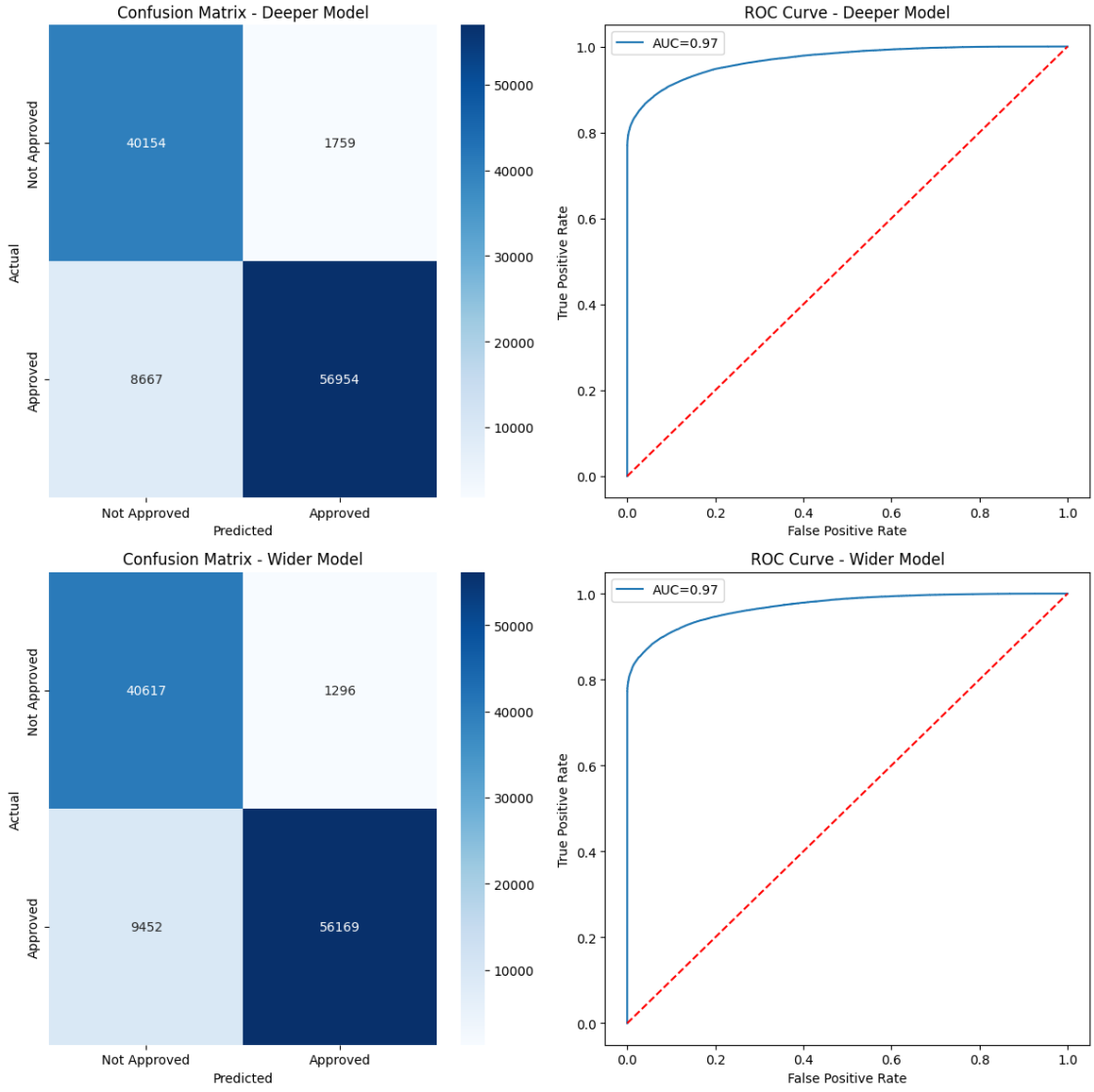
```
    plt.tight_layout()
    plt.show()

    # print final accuracy comparison
    print("Final Model Accuracy Comparison:")
    for name, acc in results.items():
        print(f"{name}: {acc:.4f}")
```

[255]:
```
# define trained models, names, and histories
trained_models = [deeper_model, wider_model]
model_names = ["Deeper Model", "Wider Model"]
histories = [deeper_hist, wider_hist]

# compare deeper and wider models side by side
compare_trained_models(trained_models, model_names, histories, X_test, y_test)
```

```
3361/3361              2s 491us/step
3361/3361              2s 687us/step
```




```
3361/3361              2s 483us/step
3361/3361              2s 452us/step
```




```
Final Model Accuracy Comparison:
Deeper Model: 0.9030
Wider Model: 0.9001
```

With our baseline model having an accuracy of 0.8874, we can conclude that the deeper model has the best performance out of the three deep learning models. Although it is still below the performance of the random forest model of 0.9155 accuracy, we will perform hyperparamater tuning in an attempt to surpass it.

[258]:
```
# hyperparameter tuning, defining the range of values
def build_model(hp):
    model = Sequential()
    model.add(Dense(hp.Int('units_1', min_value=64, max_value=256, step=64),␣
 ↪activation='relu',
```

```
                input_shape=(X_train.shape[1],)))
    model.add(Dropout(hp.Float('dropout_1', min_value=0.3, max_value=0.7,␣
 ↪step=0.1)))

    model.add(Dense(hp.Int('units_2', min_value=32, max_value=128, step=32),␣
 ↪activation='relu'))
    model.add(Dropout(hp.Float('dropout_2', min_value=0.3, max_value=0.7,␣
 ↪step=0.1)))

    model.add(Dense(hp.Int('units_3', min_value=16, max_value=64, step=16),␣
 ↪activation='relu'))

    model.add(Dense(1, activation='sigmoid'))

    model.compile(optimizer=tf.keras.optimizers.Adam(
        hp.Choice('learning_rate', [1e-2, 1e-3, 1e-4, 1e-5])
    ), loss='binary_crossentropy', metrics=['accuracy'])

    return model

# set up tuner
tuner = kt.RandomSearch(
    build_model,
    objective='val_accuracy',
    max_trials=5,
    executions_per_trial=1,
    directory='hyperparam_tuning',
    project_name='credit_approval_tuning'
)

# run tuning
tuner.search(X_train, y_train,
             epochs=10,
             validation_split=0.2,
             batch_size=32,
             class_weight=class_weights_dict,
             verbose=1)

# get the best model
best_model = tuner.get_best_models(num_models=1)[0]
```

```
Trial 5 Complete [00h 02m 23s]
val_accuracy: 0.9027630686759949

Best val_accuracy So Far: 0.9027630686759949
Total elapsed time: 00h 11m 52s

c:\Users\Admin\anaconda3\Lib\site-packages\keras\src\layers\core\dense.py:87:
```

```
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
c:\Users\Admin\anaconda3\Lib\site-packages\keras\src\saving\saving_lib.py:757:
UserWarning: Skipping variable loading for optimizer 'adam', because it has 2
variables whereas the saved optimizer has 18 variables.
  saveable.load_own_variables(weights_store.get(inner_path))
```

With the best hyperparameters discovered, we create a best model, retrain and evaluate it on all training data.

[260]:
```python
# get the best model hyperparameters from tuning
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]

print("\nBest Hyperparameters Found:")
print(f"Units 1: {best_hps.get('units_1')}")
print(f"Units 2: {best_hps.get('units_2')}")
print(f"Units 3: {best_hps.get('units_3')}")
print(f"Dropout 1: {best_hps.get('dropout_1')}")
print(f"Dropout 2: {best_hps.get('dropout_2')}")
print(f"Learning Rate: {best_hps.get('learning_rate')}")

# build best model and train
best_model = tuner.hypermodel.build(best_hps)

best_model.fit(
    X_train, y_train,
    epochs=20,
    batch_size=32,
    validation_split=0.2,
    class_weight=class_weights_dict,
    verbose=1
)

# evaluate best model on all training data
test_loss, test_acc = best_model.evaluate(X_test, y_test)
print(f"Best Model Test Accuracy: {test_acc:.4f}")
```

```
Best Hyperparameters Found:
Units 1: 256
Units 2: 128
Units 3: 16
Dropout 1: 0.4
Dropout 2: 0.4
Learning Rate: 0.0001
Epoch 1/20
10754/10754             16s 1ms/step
```

```
- accuracy: 0.8115 - loss: 0.3877 - val_accuracy: 0.8782 - val_loss: 0.2551
Epoch 2/20
10754/10754            15s 1ms/step
- accuracy: 0.8717 - loss: 0.2608 - val_accuracy: 0.8874 - val_loss: 0.2348
Epoch 3/20
10754/10754            14s 1ms/step
- accuracy: 0.8816 - loss: 0.2382 - val_accuracy: 0.8928 - val_loss: 0.2259
Epoch 4/20
10754/10754            16s 2ms/step
- accuracy: 0.8881 - loss: 0.2280 - val_accuracy: 0.8963 - val_loss: 0.2219
Epoch 5/20
10754/10754            15s 1ms/step
- accuracy: 0.8932 - loss: 0.2199 - val_accuracy: 0.8984 - val_loss: 0.2185
Epoch 6/20
10754/10754            15s 1ms/step
- accuracy: 0.8962 - loss: 0.2154 - val_accuracy: 0.8991 - val_loss: 0.2167
Epoch 7/20
10754/10754            15s 1ms/step
- accuracy: 0.8978 - loss: 0.2121 - val_accuracy: 0.9023 - val_loss: 0.2132
Epoch 8/20
10754/10754            14s 1ms/step
- accuracy: 0.8998 - loss: 0.2094 - val_accuracy: 0.9022 - val_loss: 0.2124
Epoch 9/20
10754/10754            13s 1ms/step
- accuracy: 0.8999 - loss: 0.2084 - val_accuracy: 0.9029 - val_loss: 0.2108
Epoch 10/20
10754/10754            14s 1ms/step
- accuracy: 0.9013 - loss: 0.2068 - val_accuracy: 0.9027 - val_loss: 0.2108
Epoch 11/20
10754/10754            13s 1ms/step
- accuracy: 0.9016 - loss: 0.2059 - val_accuracy: 0.9033 - val_loss: 0.2101
Epoch 12/20
10754/10754            14s 1ms/step
- accuracy: 0.9016 - loss: 0.2047 - val_accuracy: 0.9040 - val_loss: 0.2091
Epoch 13/20
10754/10754            14s 1ms/step
- accuracy: 0.9028 - loss: 0.2037 - val_accuracy: 0.9044 - val_loss: 0.2078
Epoch 14/20
10754/10754            13s 1ms/step
- accuracy: 0.9028 - loss: 0.2030 - val_accuracy: 0.9044 - val_loss: 0.2068
Epoch 15/20
10754/10754            15s 1ms/step
- accuracy: 0.9030 - loss: 0.2020 - val_accuracy: 0.9058 - val_loss: 0.2054
Epoch 16/20
10754/10754            14s 1ms/step
- accuracy: 0.9036 - loss: 0.2014 - val_accuracy: 0.9057 - val_loss: 0.2050
Epoch 17/20
10754/10754            15s 1ms/step
```

```
- accuracy: 0.9045 - loss: 0.2003 - val_accuracy: 0.9060 - val_loss: 0.2052
Epoch 18/20
10754/10754             15s 1ms/step
- accuracy: 0.9041 - loss: 0.2002 - val_accuracy: 0.9064 - val_loss: 0.2039
Epoch 19/20
10754/10754             14s 1ms/step
- accuracy: 0.9046 - loss: 0.1994 - val_accuracy: 0.9056 - val_loss: 0.2046
Epoch 20/20
10754/10754             13s 1ms/step
- accuracy: 0.9045 - loss: 0.1990 - val_accuracy: 0.9065 - val_loss: 0.2040
3361/3361             2s 710us/step -
accuracy: 0.9046 - loss: 0.2083
Best Model Test Accuracy: 0.9056
```

7 Analysis of results

The performance of the various deep learning models we have created and covered, including the baseline, wider, deeper, and best hyperparameter tuned model will be presented and summarised below. The evaluations are based on key metrics such as accuracy, f1 score, and AUC(area under curve), which help to uncover each model's effectiveness in classifying credit approval applications.

Baseline model

- The baseline model, as above, has achieved an accuracy score of 0.8874. This indicates reasonable predictive capability. It also has an AUC of 0.97, which shows that it is able to distinguish between classes reasonably well, as ideal models have an AUC of over 0.95. However, the model's validation loss stops improving while training loss continues to decrease ever so slightly, which indicates that training the model to more epochs might lead to potential overfitting.
- From the confusion matrix, we are able to tell that the model is able to correctly predict a majority of both classes. However, there are alot more false negatives (11288) to false positives (810), which tells me that this model has a higher accuracy in not approving applicants than approving.

Wider model

- The wider model, has more neurons in the same number of layers as the baseline model, which in theory should help it capture complex relationships better than the baseline model. Validation accuracy was consistently higher than training accuracy in early epochs, indicating faster learning compared to the deeper model. The model was evaluated to an accuracy score of 0.9001, which is slightly higher than the baseline model. While it did perform marginally better as compared to the baseline model, the increase in more neurons per layer does not drastically improve predictive power.
- Although the AUC score is the same as the other models, the confusion matrix shows that it retains more balanced discrimination capabilities. With false negatives numbering 9452 (as compared to the baseline's 11288) and false positives of 1296 (against baseline's 810), we can

tell that this model will make fewer false approvals in exchange for more eligible applicants being falsely rejected.

- In general, the wider model learns faster in early epochs and generalises well, making it the top choice for training on large datasets.

Deeper model

- The deeper model, has more hidden layers than the baseline model, exploring if depth improves learning. It achieved an accuracy of 0.9030, outperforming both baseline and wider models. However, only minimally, indicating that dataset complexity may already be sufficiently captured by the baseline structure.
- The model's training loss decreased smoothly, and validation loss remained stable, suggesting good generalisation. From the confusion matrix, it's false negattives, number 8667, and false positives, number 1759, making this model the more balanced model out of the others.

Hyperparameter tuned model

- This model outperformed all 3 other deep learning models, with an accuracy score of 0.9056. The validation accuracy remains stable and close to the training accuracy, showing minimal overfitting. Lower dropout rates, at 0.4, showed improved generalization, without overfitting.

Overall analysis

- If the company's priority is to minimise financial risk, the baseline model would be the top pick, with the lowest number of false positives, 810.
- If the company's priorty is to balance fair approvals and financial risk, the hyper-tuned model would be recommended, with its best overall generalistion and performance.
- If the company's priority is fair approvals, the deeper model would be the right pick, with the least number of false negatives, at 8667.
- However, the traditional machine learing approach of random forest still has the highest accuracy scores of 0.9155, making it the best model if we are only looking at accuracy scores.
- Although not definitive, this could also prove that the random forest method may be more suitable for this dataset than deep neural networks.

8 Conclusion

In this report, we developed and explored various machine learning and deep learning models to predict credit card approvals. We focused on handling class imbalance, optimising architectures, and improving generalisation. The models were evaluated based on accuracy, AUC, confusion matrix for false positive and false negative rates, providing insight into how different architectures balance risk management and fairness.

The baseline model was the most conservative, prioritising risk minimisation by rejecting more applicants to reduce false approvals. The deeper and wider models improved accuracy every so slightly, but did not drastically outperform the baseline. The hyperparameter tuned model provided the best balance, with its best overal generalisation and highest accuracy (out of deep learning models).

Among the traditional models, random forest performed best with the highest accuracy out of all the models explored. While logistic reggresion performed sub optimally as compared to the rest of the models.

In conclusion, the choice of model depends on the company's goal or priority, with the baseline model, hyperparameter tuned model, and the deeper model being the top picks out of the deep learning models. And the random forest model, out of the machine learning models, for it's accuracy, speed, and simplicity.

9 Future Work

While the models presented promising results, there are a few areas that can still be explored further.

1. Robust validation techniques
    - Implementing k-fold cross-validation to ensure that the model's performance generalizes well across different subsets of data.
    - Conducting additional stress tests on imbalanced data to assess real-world reliability.
2. Improving model explainability
    - Using techniques like SHAP (Shapley Additive Explanations) and LIME (Local Interpretable Model-agnostic Explanations) to improve trust and transparency in predictions.
    - Ensuring that model decisions align with regulatory and ethical considerations in credit approvals.

10 References

1. Chollet, F. (2018). Deep learning with Python. Manning Publications
2. MoneyMan. (2020, March 24). Credit card approval prediction. Kaggle. https://www.kaggle.com/datasets/rikdifos/credit-card-approval-prediction