Keras Sequential Model Architecture

YU-CHIEH HSU

August, 2025

1 Introduction

This report outlines the architecture, tuning strategies, and evaluation of the Churn Prediction Artificial Neural Network. The following section explains the purpose of adopting Keras as the modeling framework and provides an overview of its structure. The next section examines the choice of the initialiser and activation functions for the layers, followed by a discussion of the optimisation process and its parameters. The report presents an additional section on fine-tuning, illustrating how it integrates with the Keras model. Finally, the report includes an evaluation of the model through TensorBoard visualisations, demonstrating how the chosen architecture and tuning decisions affected training and validation outcomes.

2 Model Structure

The development of this model was informed by *Hands-on Machine Learning with Scikit-Learn, Keras and TensorFlow* as well as supplementary community discussions and online resources. The Keras deep learning API is widely adopted due to its simplicity and efficiency in building various types of neural networks. Moreover, it is compatible with multiple deep learning frameworks, including TensorFlow, which was subsequently used in this project for evaluation with TensorBoard. Below is the final model structure written in Python.

```
def build_model(input_dim:int, units1:int, units2:int, lr:float, dropout:float,
l2:float=1e-4):
    # Initialise ANN
    model = Sequential()

    # Input layer and 1st hidden layer
    model.add(Dense(units1, kernel_initializer='he_normal',
    kernel_regularizer=reg.l2(l2), use_bias=False, input_shape=(input_dim,)))
    model.add(BatchNormalization())
    model.add(tf.keras.layers.Activation('relu'))
    model.add(Dropout(dropout))

    # 2nd layer
    model.add(Dense(units2, kernel_initializer='he_normal',
    kernel_regularizer=reg.l2(l2), use_bias=False))
    model.add(BatchNormalization())
    model.add(tf.keras.layers.Activation('relu'))

    # Output layer
    model.add(Dense(1, kernel_initializer='glorot_uniform',
    activation='sigmoid'))

    # Optimise
    model.compile(optimizer=Adam(learning_rate=lr), loss='binary_crossentropy',
    metrics=[tf.keras.metrics.AUC(curve='PR', name='auprc'), 'accuracy'])

    return model
```

Keras provides a layer breakdown that shows each layer in the Sequential model, the output dimensions, and the parameters (weights and biases) the layer contains (Fig.1). The bottom of the model summary includes the total trainable, non-trainable, and optimiser parameters. The architecture combines dense layers with batch normalisation, activation, and dropout. The final layer outputs a single probability for churn prediction.

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 32)	352
batch_normalization (BatchNormalization)	(None, 32)	128
activation (Activation)	(None, 32)	0
dropout (Dropout)	(None, 32)	0
dense_1 (Dense)	(None, 64)	2,048
batch_normalization_1 (BatchNormalization)	(None, 64)	256
activation_1 (Activation)	(None, 64)	0
dense_2 (Dense)	(None, 1)	65

Total params: 8,165 (31.90 KB)

Trainable params: 2,657 (10.38 KB)

Non-trainable params: 192 (768.00 B)

Optimizer params: 5,316 (20.77 KB)

Figure 1: The model summary implemented in Keras (TensorFlow backend).

Another visualisation, generated through TensorBoard, presents the sequential architecture of the model (Fig.2). This graph is often used for demonstrating transparency and debugging evidence.

3 Weight Initialiser, Activation, and Compile

1. **Input and Hidden Layers**: He normal + ReLU
2. **Output Layer**: Glorot uniform + Sigmoid
3. **Optimiser**: Adam
4. **Loss function**: Binary cross-entropy
5. **Metrics**: AUPRC and accuracy

The model was compiled with accuracy as the primary performance metrics, while the area under the precision-recall curve was also monitored to account for class imbalance.

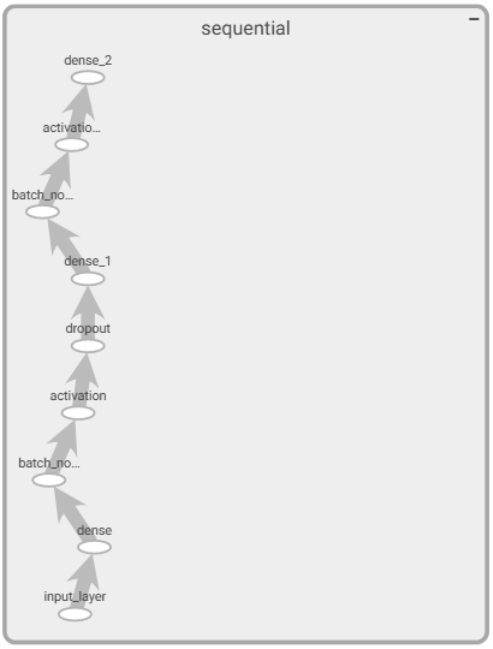

Figure 2: A visual representation of the Sequential pipeline. It confirms that the model has been correctly constructed and allows for easy inspection of layer connectivity.

4 Tuning

This report adopted Hyperband for hyperparameter optimisation, as it efficiently eliminates poor parameter choices in early trials. Although slower than RandomSearch in some cases, Hyperband explores the search space more deeply, often yielding superior configurations for retraining. To ensure robustness and avoid overfitting during searches, early stopping is usually incorporated. The same module provides two functions: one defines the search space for hyperparameters and the other creates a tuner object that runs the hyperparameter search.

```
def hypermodel(hp, input_dim:int):
    u1 = hp.Int('units1', min_value=32, max_value=128, step=32)
    u2 = hp.Int('units2', min_value=16, max_value=64, step=16)
    lr = hp.Float('lr', 2e-4, 2e-3, sampling='log')
    dr = hp.Float('dropout', 0.2, 0.4, step=0.1)
    return build_model(input_dim, u1, u2, lr, dr)
```

The maximum epochs and executions per trial were altered to achieve an average tuning time of approximately 30 minutes, resulting with an AUPRC of 0.6 and a test accuracy of around 0.78.

```
def make_tuner(input_dim:int, project_name='krs_hyperband',
directory='hyperband'):
    tuner = kt.Hyperband(
        hypermodel=lambda hp:hypermodel(hp, input_dim),
        max_epochs=30,
        factor=3,
        hyperband_iterations=1,
        objective=Objective('val_auprc', direction='max'),
        executions_per_trial=3,
        directory=directory,
        project_name=project_name,
        overwrite=True)
    return tuner
```

5 Evaluation

The evaluation focuses on multiple metrics to ensure a balanced assessment of the model performance. TensorBoard visualisations, including AUPRC and loss curves, are used to track the learning dynamics and identify potential overfitting (Fig.3-4). A kernel distribution that monitors the learning process by detecting the weight changes over epochs (Fig.5). In addition, a confusion matrix was generated to provide a clearer view of class-level performance (Fig.6).

This model also adopted an optional fitting using Gradient Boosting with calibrated cross-validation. While this result produced a slightly lower overall accuracy, it offered a more reliable confusion matrix, reflecting improved handling of minority churn cases.

Besides the model architecture and tuning function where generalisations were applied to avoid overfitting, the model fitting process also includes a callback to stop training as soon as the validation metric stops improving.

```
callbacks = [tf.keras.callbacks.EarlyStopping(monitor='val_auprc',
mode='max', patience=2, min_delta=1e-4, restore_best_weights=True),
ReduceLROnPlateau(monitor='val_auprc', mode='max', factor=0.5, patience=2,
verbose=0)]
```

`restore_best_weights` return to the weights that gave the best validation performance. The `ReduceLROnPlateau` reduces the learning rate when the model stops improving.

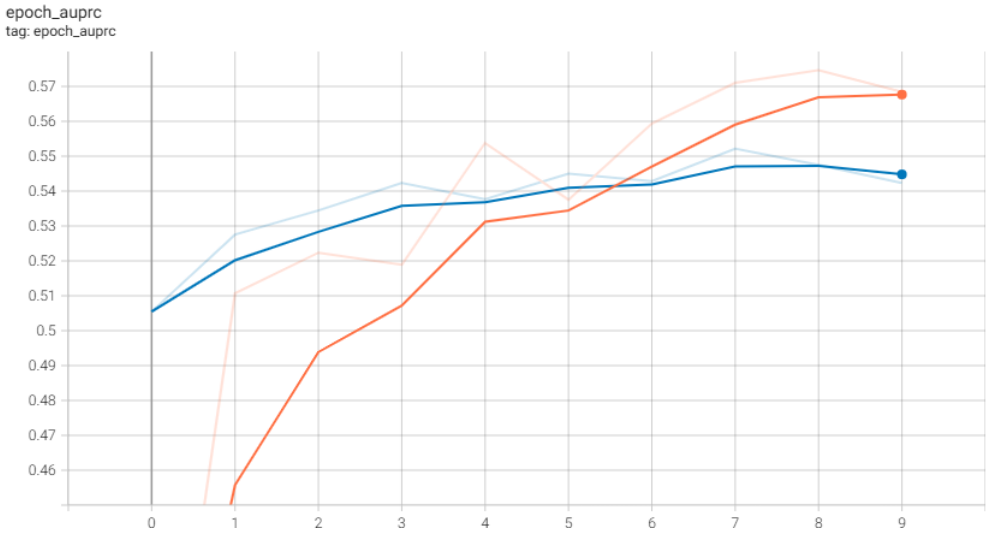

Figure 3: AUPRC curves over training epochs for both training(orange) and validation(blue) sets. Both curves rise steadily, and the validation curve closely follows the training curve, suggesting that the model is improving its ability to distinguish churn cases.

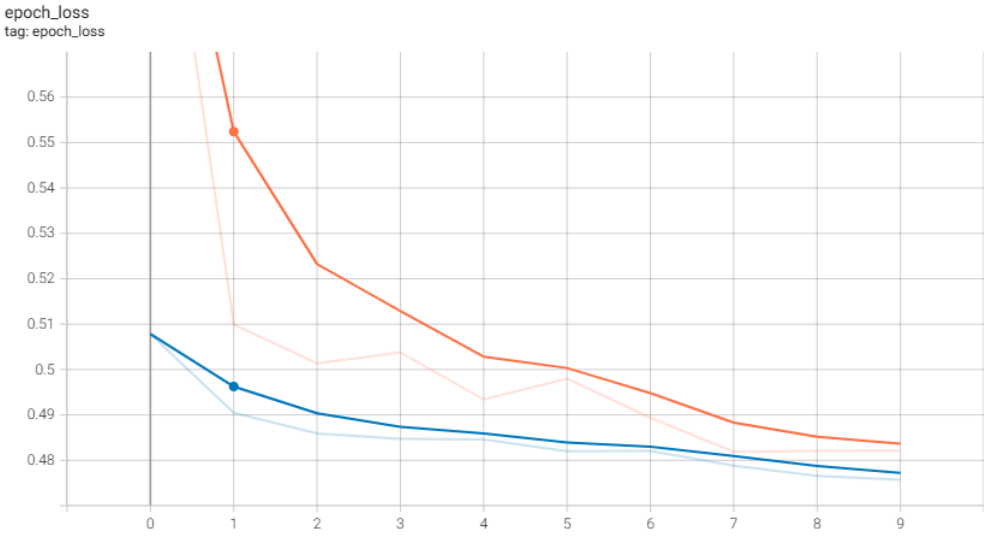

Figure 4: Training and validation loss curves across epochs. Both losses decrease consistently, with no sharp divergence, indicating stable learning and no significant overfitting. The gap might imply a strong regularisation or dropout exists in the layers.

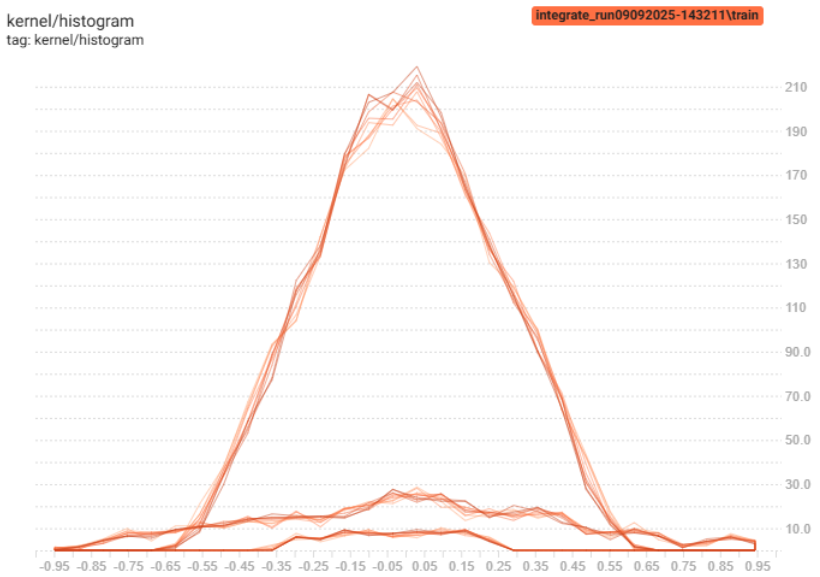

Figure 5: A bell-shaped kernel distribution indicates that the model has learn effectively over epochs and not dependent on specific weights.

To compare the two confusion matrices, model scores were converted to calibrated probabilities on the validation set, then passed to the precision-recall curve to select the operating threshold that maximises either accuracy or F1 on validation; the fixed threshold was then applied to the test set for final reporting. F1 score balances precision and recall, especially when dealing with imbalanced data (churners : non-churners $\approx 27 : 73$). By maximising F1, the threshold is more likely to balance catching as many churners as possible without making too many false alarms.

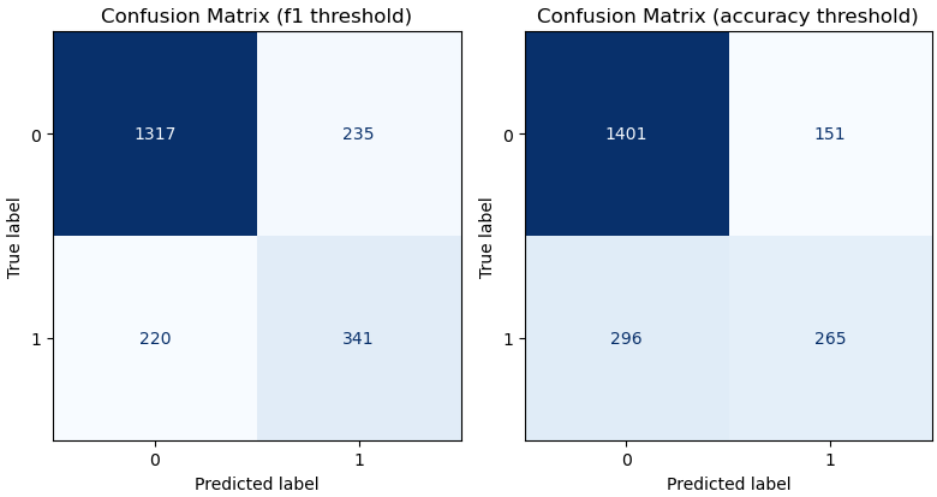

Figure 6: Confusion matrix calculated based on different thresholds: F1 score (left) and accuracy/default (right). The left matrix balances precision and recall more evenly, resulting in a higher number of correctly predicted labels overall.

6 Summary

This report documents the design, training, tuning, and evaluation of a Keras-based Artificial Neural Network for customer churn predictions. The Keras ANN, tuned via Hyperband and monitored with TensorBoard, produced robust churn predictions with stable learning and minimal overfitting. AUPRC was selected because the given dataset involves imbalanced churner and non-churner rate, and the F1 threshold approach is recommended for deployment, as it produced more correctly predicted churners, making it the preferred choice for this task.