# Customer Churn Prediction Using Artificial Neural Networks (ANN)

## Summary

This report documents a complete, end-to-end churn prediction pipeline built on the **Churn Modelling** dataset. The goal is to predict whether a retail bank customer will **exit** in the near term. The following is the process:

1. Explore the data to identify patterns (EDA)
2. Engineer features and apply encoders/scalers
3. Train a regularized **ANN** with early stopping and visualize learning with TensorBoard
4. Evaluate using confusion matrix, Precision/Recall/F1, and ROC–AUC
5. Tune the classification threshold to maximize F1 and improve recall
6. Ship a single-customer scoring pipeline

The final model achieves 87.1% accuracy and an AUC of around 0.87 on the hold-out set. At the default 0.50 threshold, it delivers high precision but only about 47% recall.

Lowering the threshold to 0.36 boosts both the F1 score and the catch-rate of churners. This trade-off makes sense when the cost of missing a churner is higher than the cost of reaching out unnecessarily.

# 1. Business Problem Formulation

- **Objective.** Predict the probability that a currently active customer will churn (binary classification).
- **Decision.** Trigger a retention action when predicted probability ≥ decision threshold .
- **Success Criteria.** High **recall** for churners while keeping **precision** acceptable, because missing a churner (FN) is typically more expensive than contacting a non-churner (FP).

**Why ANN?**

I chose an ANN because the tabular data has complex, non-linear interactions between features (for example, age × balance × geography). Using ReLU activations with regularization allows the model to capture these relationships. This serves as a strong baseline (later improvements can include comparisons with tree-based models as well).

# 2. Data Overview

**Source:** Churn_Modelling.csv (10,000 customers; typical Kaggle/UCI-style dataset)

**Target:** Exited — 1 for churned, 0 for retained

**Schema overview:** Demographics (Age, Gender, Geography), account details (Balance, NumOfProducts, HasCrCard, IsActiveMember, EstimatedSalary), and tenure metrics (Tenure, CreditScore).

**Removed identifiers:** RowNumber, CustomerId, Surname (these contain no behavioral information and can lead the model to memorize individual customers instead of learning patterns that generalize).

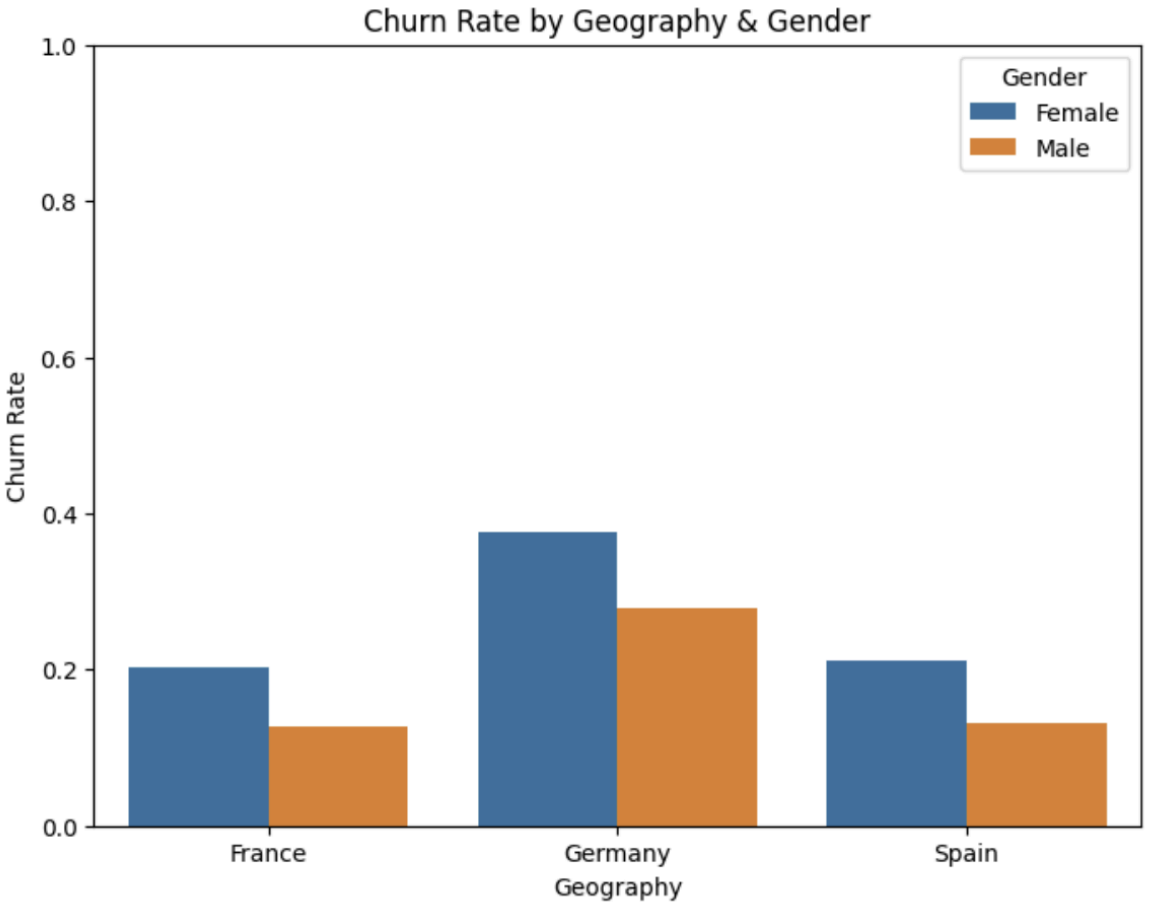
# 3. Exploratory Data Analysis (EDA)

Targeted visualizations were used to explore distributions and churn relationships

## 3.1 Churn Rate by Geography & Gender

**Question:** Does churn vary by region, and does gender interact with geography?

**Findings:**

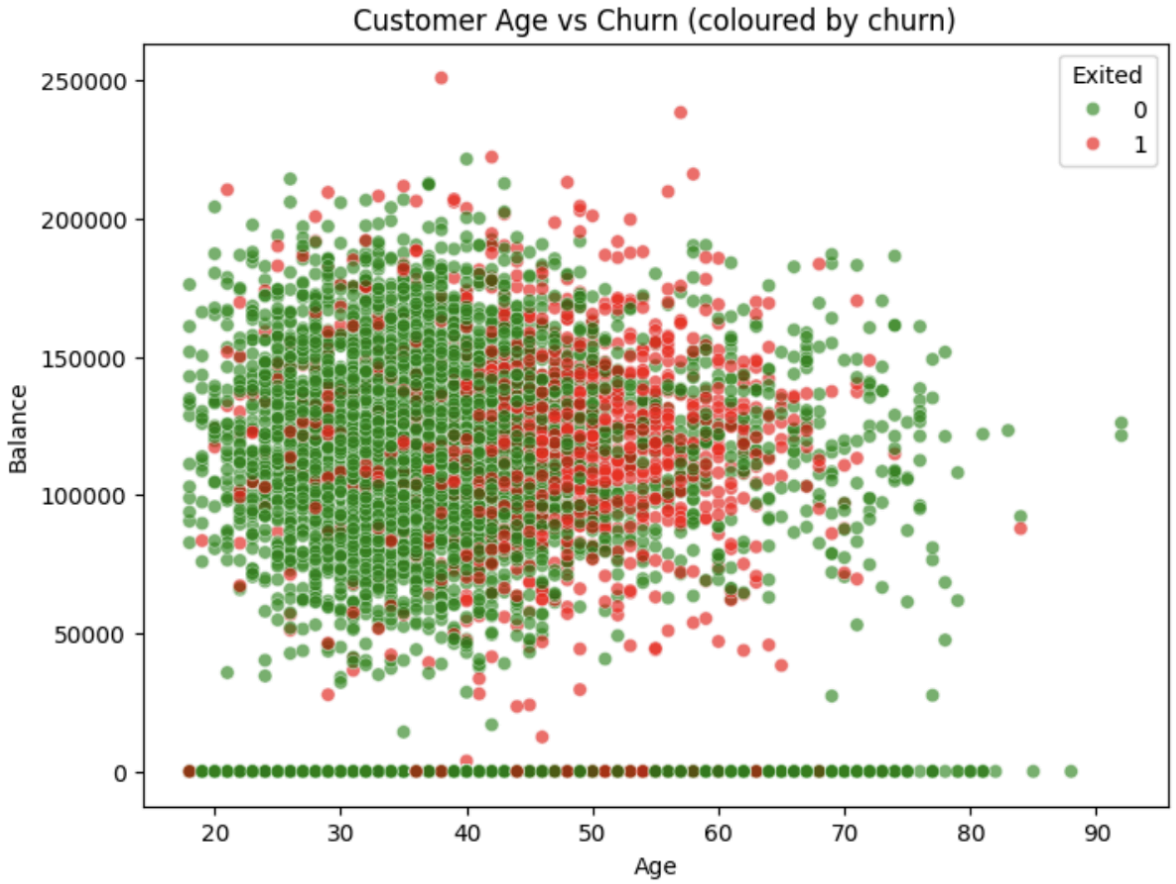



- Germany has the highest churn rate, with females particularly elevated (~40%).
- France and Spain show lower churn for both genders.

**Implication:** Prioritize German females, followed by German males, in retention efforts.

## 3.2 Customer Age vs Churn (colored) vs Balance

**Question:** Is churn concentrated in certain age or balance ranges?



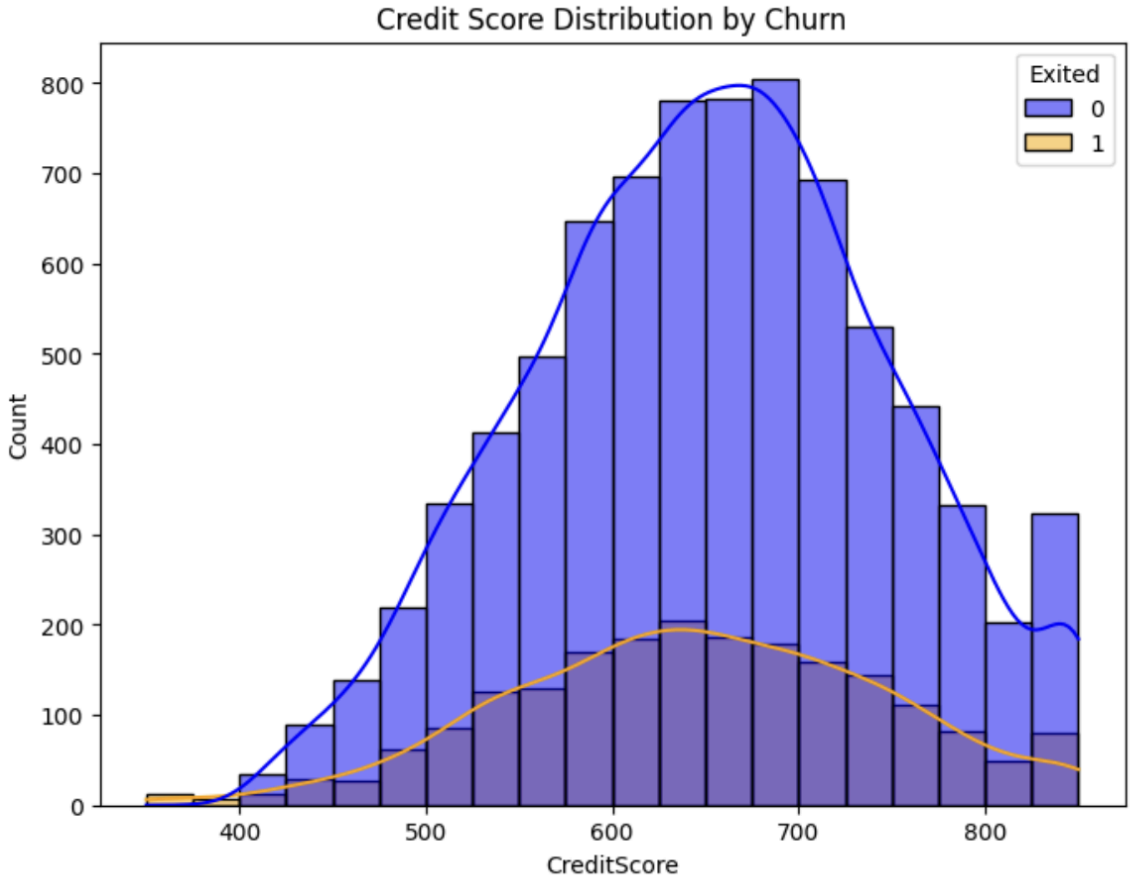

**Findings:**

- Churners are more common in the middle-age group (~40–60).
- High balances (>100k) don’t necessarily protect against churn.

**Implication:** Monetary value alone isn’t a safeguard; other factors drive churn.

### 3.3 Credit Score Distribution by Churn

**Question:** Can credit quality separate churners from non-churners?



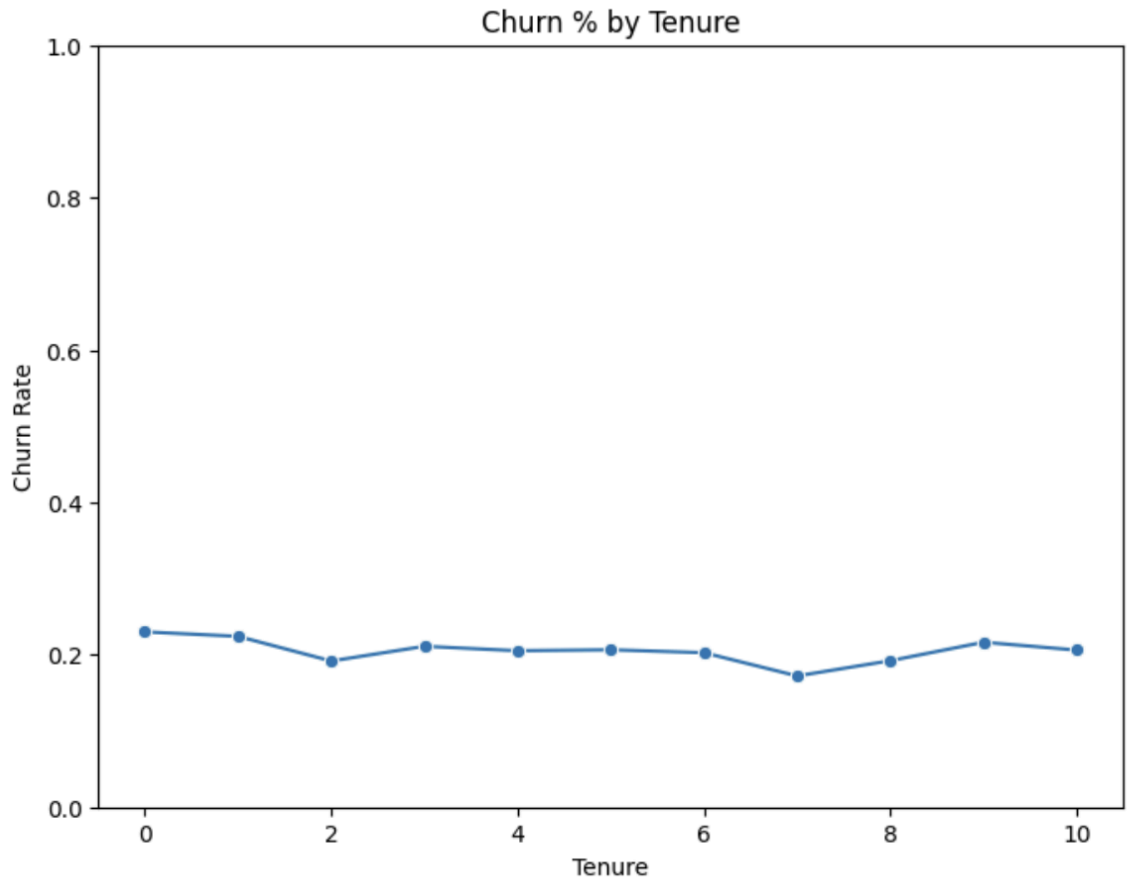

**Findings:**

- Non-churners skew slightly higher in credit score, but distributions heavily overlap.

**Implication:** Credit score has some predictive value, but requires combination with other variables.

## 3.4 Churn % by Tenure

**Question:** Does tenure meaningfully reduce churn?

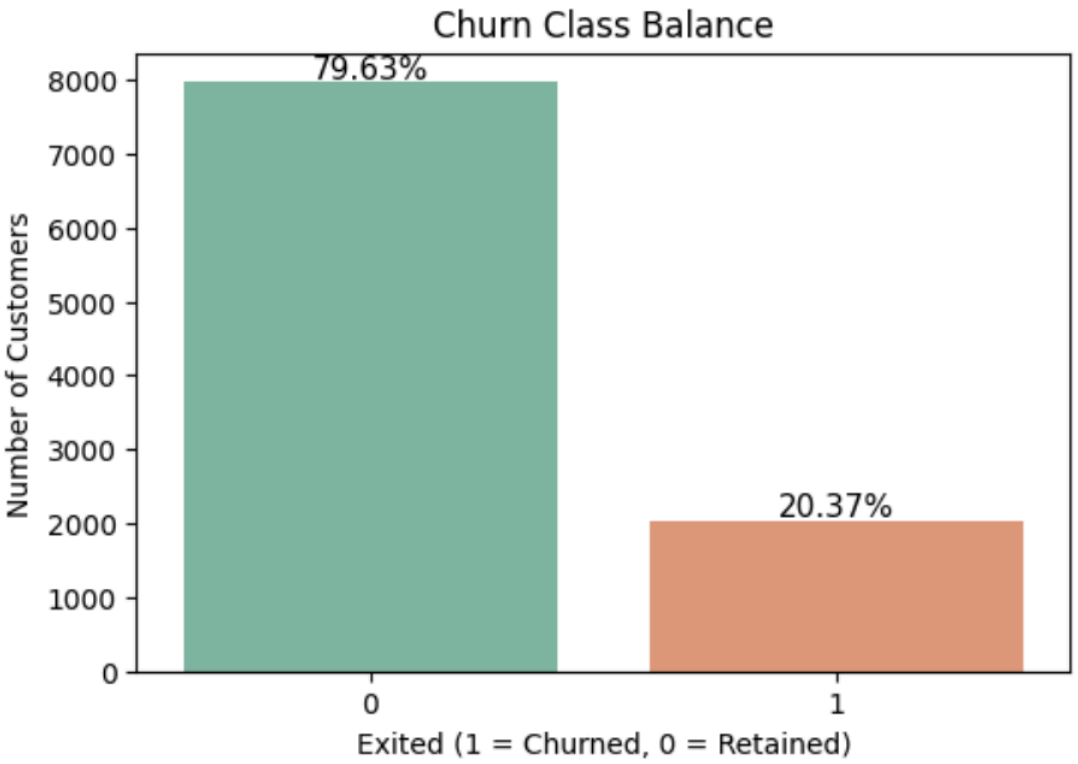



**Observations:** Churn rate is relatively **flat** across tenure (≈0.17–0.24).

**Implication:** Tenure provides limited standalone signal in this dataset

## 3.5 Churn Class Balance

**Question:** What is the overall distribution of churn vs retention in the dataset?




**Findings:**

- **79.63%** of customers are retained (Exited = 0).
- **20.37%** of customers have churned (Exited = 1).
- The dataset is **imbalanced**, with churners being the minority class

**Implication:**

- Current analysis proceeds without balancing, but techniques such as oversampling (SMOTE), undersampling, or class weights could be applied during model training to address the imbalance and improve recall for churners.

1. **Oversampling (e.g., SMOTE)**

   **Description:** Increases the number of minority-class samples by duplicating or synthesizing new ones. SMOTE (Synthetic Minority Oversampling Technique) creates synthetic churner examples by interpolating between real samples.

   **Pros:** Gives the model more churn examples to learn from.
   **Cons:** Risk of overfitting if oversampling is excessive.

   **Example (with imbalanced-learn library):**

   from imblearn.over_sampling import SMOTE

```
smote = SMOTE(random_state=42)

X_resampled, y_resampled = smote.fit_resample(X_train, y_train)

print("Before:", y_train.value_counts())

print("After:", y_resampled.value_counts())
```

2. **Undersampling**

   **Description:** Reduces the majority class (retained customers) so its size matches the minority class.
   **Pros:** Simple, avoids synthetic data.
   **Cons:** Discards useful data, which can hurt performance.

   **Example:**

```
from imblearn.under_sampling import RandomUnderSampler

rus = RandomUnderSampler(random_state=42)

X_resampled, y_resampled = rus.fit_resample(X_train, y_train)

print("Before:", y_train.value_counts())

print("After:", y_resampled.value_counts())
```

3. **Class Weights**

   **Description:** Increases the loss penalty for misclassifying minority-class samples, without changing the dataset.
   **Pros:** Keeps all data, easy to implement in most ML frameworks.
   **Cons:** Needs careful tuning to avoid instability.

**Example (Keras):**

```
class_weight = {0: 1.0, 1: 4.0}  # Heavier weight for churners

model.fit(X_train, y_train, class_weight=class_weight, epochs=100, batch_size=32)
```

# 4. Data Preparation

## 4.1 Train/Test Split

- **Train/Test Split:** 80/20 with random_state=40 for reproducibility.
- **Rationale:** An 80/20 split gives sufficient data to train the ANN effectively while reserving a sizable hold-out set for validation.

## 4.2 Encoding Strategy

**Gender:** Encoded with LabelEncoder (binary: {0, 1}).
**Why:** As a binary variable, label encoding is sufficient and avoids adding unnecessary dimensions.

**Geography:** One-hot encoded into Geography_France, Geography_Germany, and Geography_Spain.
**Why:** For nominal multi-class variables, one-hot encoding avoids the artificial ordering (like 0,1,2) that label encoding would introduce, allowing the model to learn independent weights for each category.

Encoders are **fitted on the training set only** and saved (via pickle) to ensure that validation and future scoring datasets use the exact same category mappings.

## 4.3 Feature Scaling

**Standardization:** Standardize all numeric features using StandardScaler, which converts each to z-scores (subtract the mean and divide by the standard deviation).

**Why for ANN:** Since neural networks are gradient-based optimizers, keeping features on a similar scale is important. Standardization:

- Makes the loss surface better structured, which helps the optimizer converge faster and more stably.
- Prevents large-scale variables (e.g., *Balance* in thousands) from overpowering smaller-scale ones (e.g., *NumOfProducts*). Without scaling, the optimizer might take overly small steps on small-scale features or overshoot on large-scale ones.
- Allows me to use a single learning rate effectively across all dimensions, rather than having to tune it to compensate for scale differences.

Scaler is fitted **only on the training set** and persist it (pickle) so it can be reused consistently in the live scoring pipeline.

**NOTE:** When applying it later, the order of features during transformation must match the training order (check using store scaler.feature_names_in_ to make sure nothing gets reordered accidentally).

# 5. Model Design

## 5.1 Architecture

Input
→ Dense(64, ReLU, L2=1e-3)
→ Dropout(0.30)

→ Dense(32, ReLU)
→ Dropout(0.30)

→ Dense(1, Sigmoid)

**Rationale.**

**ReLU activation**: Defined as f(x) = max(0, x), ReLU outputs zero for negative inputs and passes positive values unchanged. This keeps gradients from shrinking towards zero (avoiding the *vanishing gradient problem*) and allows for faster convergence compared to saturating activations like sigmoid/tanh in hidden layers. ReLU also introduces sparsity (many neurons will output exactly zero) which can act as an implicit form of regularization.

**Width 64 to 32**: The first dense layer with 64 units captures a high-dimensional representation of the input feature space, which allows the model to learn a wide variety of non-linear interactions. The second dense layer with 32 units forces the network to retain only the most informative features.

- **L2 regularization (λ = 1e-3)**: Applied to the kernel weights of the first dense layer.

  The loss function becomes:

$$L_{total} = L_{binary\_crossentropy} + \lambda \sum w^2$$

- This reduces the affect  of large weight magnitudes, helping to maintain stability especially on unseen data.

- **Dropout (0.30)**: Randomly sets 30% of activations in a layer to zero during each training step. This breaks up dependency between neurons and improves generalization. At inference time, all neurons are active but outputs are scaled to account for the dropout during training.

- **Sigmoid output**: This squashes the output into a [0, 1] range, which we use as the probability of churn. This aligns with binary classification.

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

So the wide first layer learns high-level combinations of features. The second, narrower layer compresses these into a signal. L2 and dropout together mitigate overfitting risks common in small-to-medium datasets. The Sigmoid output directly yields a churn probability for threshold based decisions.

## 5.2 Optimization & Training Controls

The training process aims to find model parameters (weights and biases) that minimize the total loss. The optimizer doesn't "know" the shape of the loss surface in advance, it explores it step by step, adjusting parameters based on gradients until it reaches a good minimum.

**Optimizer – Adam**

- **What it is:** Adam (Adaptive Moment Estimation) combines the benefits of two other optimizers:
  - Momentum: keeps track of an exponentially decaying average of past gradients to smooth out noisy updates.
  - RMSProp → scales learning rates for each parameter individually based on the magnitude of past gradients.
- **Why used here:** Adam adapts step sizes per parameter which works well with sparse gradients from ReLU activations. A smaller learning rate is used here to avoid overshooting the optimum.

**Loss Function – Binary Cross-Entropy**

$$L_{\text{BCE}} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(p_i) + (1 - y_i)\log(1 - p_i)]$$

- **Why used here:**
  - A good choice for binary classification (churn / no churn).
  - Treats the output of the final Sigmoid as a probability.
  - Matches the Bernoulli likelihood assumption.

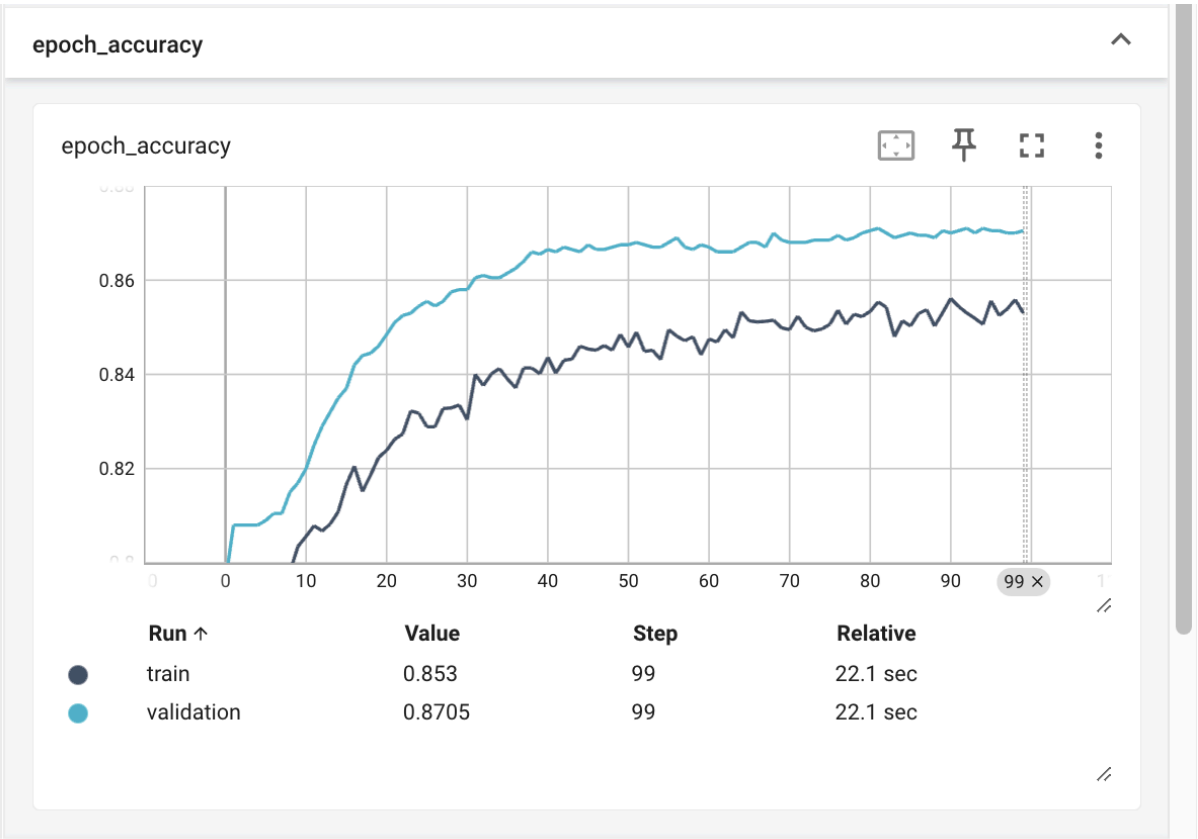
**Callbacks**

1. **EarlyStopping**
    - **Settings:** patience=5, monitor='val_loss', restore_best_weights=True
    - **Purpose:** Stops training when the validation loss hasn't improved for 5 consecutive epochs. This avoids overfitting (training too long and memorizing noise) and saves computation time. It also restores the best weights.
2. **TensorBoard**
    - **Settings:** histogram_freq=1
    - **Purpose:** Logs training and validation metrics, activation histograms, and weight distributions for each epoch.

# 6. Training Diagnostics (TensorBoard)

## 6.1 Accuracy Curves

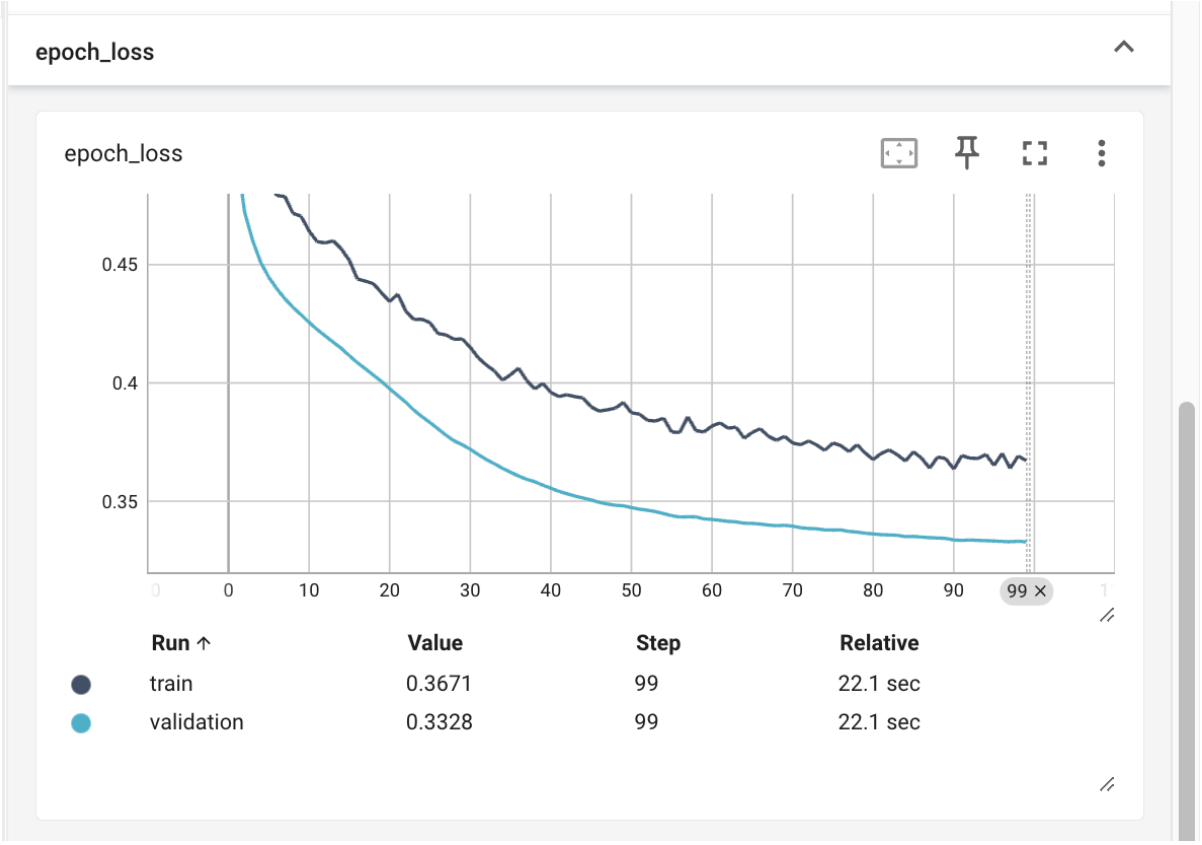
- **Pattern:** Validation accuracy climbs and stabilizes around **0.87**
- **Why this is good:** Generalization exceeds training: the model is **not memorizing** and benefits from regularization + early stopping.




## 6.2 Loss Curves

- **Pattern:** Both training and validation loss decline smoothly

- **Interpretation:** Optimization is **stable**; no signs of dramatic overfitting.




# 7. Evaluation on Hold-out Set

## 7.1 Confusion Matrix (t = 0.50)

**Counts:**

- **TN:** 1,559
- **FP:** 57
- **FN:** 202
- **TP:** 182
- **Total:** N=2000N=2000

**Metrics:**

- **Accuracy:** 0.8705
- **Precision (Churn):** 0.7615
- **Recall (Churn):** 0.4740
- **F1-score:** 0.5843

**Insights:**

- **False Negatives (202):** Actual churners the model missed. That might lead to no retention action taken, leading to potential revenue loss.
- **False Positives (57):** Customers incorrectly flagged as churners. These would be unnecessary retention contacts. Could lead to minor cost and possible customer annoyance.
- **Action point:** In most churn scenarios, **recall** (catching as many churners as possible) is more important

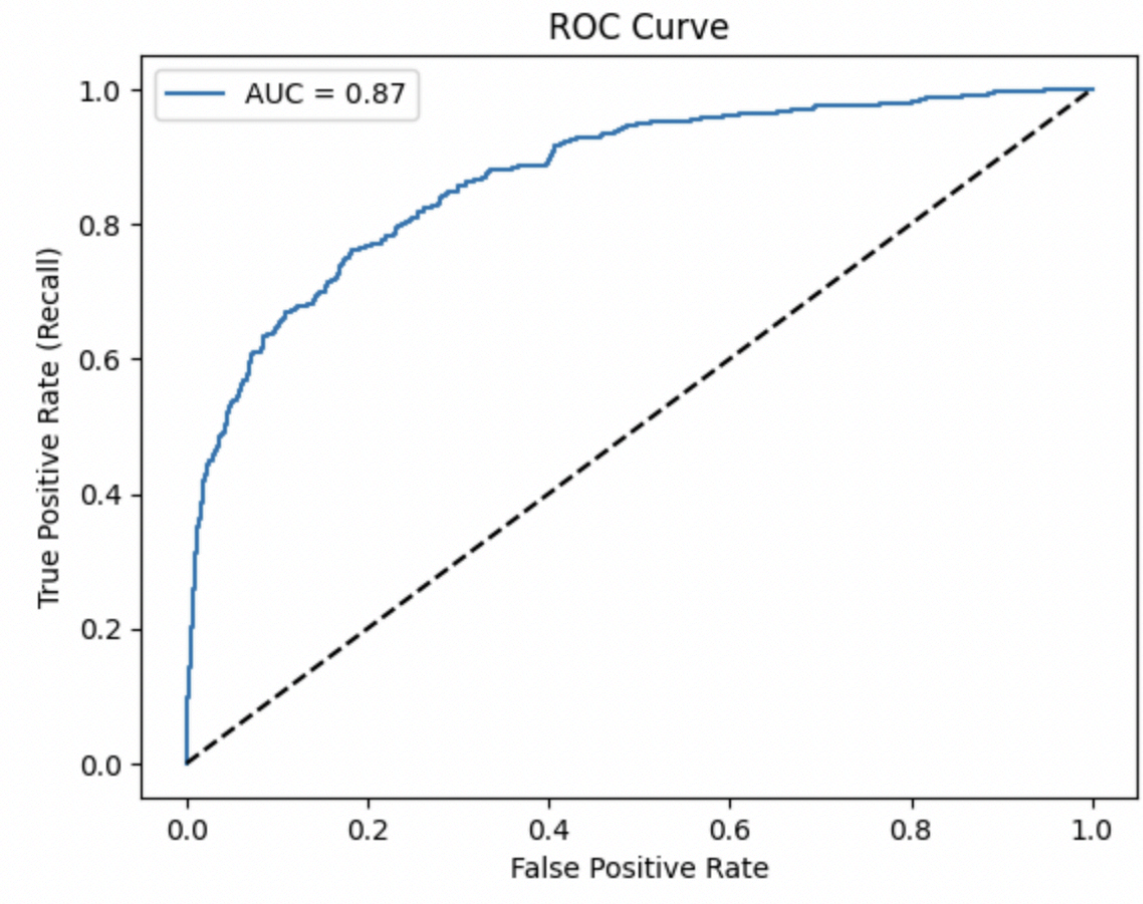
## 7.2 ROC Curve & AUC

AUC ≈ 0.87 means that if we pick one churner and one non-churner at random, the model will give the churner a higher churn score about 87% of the time. We use ROC because it measures this ranking ability across all possible thresholds, not just one.

Even though churners are only 20% of the data, the model has learned patterns that often put churners above non-churners in the ranking, which is why the AUC is high.

However, recall is low (0.47) because at the default cutoff (0.5), many churners' scores still fall below the threshold and get classified as "stay."

For example: if a real churner gets a score of 0.48 and a non-churner gets 0.30, the churner is ranked correctly (good for AUC) but is still predicted as "stay" (bad for recall).




## 8. Decision Threshold Tuning

The model outputs a probability for each customer, e.g., 0.82 means “82% chance of churn.”

- At the default threshold of **0.50**, anyone with p≥0.50 is predicted as “will churn,” otherwise “will stay.”
- But this default isn’t always optimal,  especially in imbalanced datasets where churners are rare (20% here).

So with fewer churn examples in training, the model learns to be cautious about predicting churn, pushing churners scores lower on average. This means many genuine churners fall just under the cutoff.

**Too high a threshold** -> Many actual churners with scores like 0.48 or 0.46 get classified as “stay” even though the model ranked them above most non-churners. This leads to false negatives

**Too low a threshold** -> We capture more churners, but also end up flagging more non-churners. This creates false positives, which might mean unnecessary retention offers or outreach (and very annoyed customers)

So I ran a **threshold sweep** from 0.00 to 1.00 in 0.01 steps on the validation set. For each threshold,

- **F1-score** = Harmonic mean of precision and recall, rewarding balance between the two.

Then picked the threshold that gave the highest F1.

- **Optimal threshold: 0.36** (higher F1 than default 0.50)

**Effect of lowering threshold from 0.50 to 0.36:**

- **Recall increases** –> we catch more churners.
  Example: A churner with a score of 0.48 is missed at 0.50 but caught at 0.36.
- **Precision decreases** –> we get more false alarms, since some non-churners with scores between 0.36 and 0.50 are now flagged.
- **F1 improves** –> in our case, the gain in recall outweighed the drop in precision, making the model's overall effectiveness better.

If the cost of missing a churner is much higher than the cost of contacting a non-churner, lowering the threshold is often worth it.

# 9. Production-Ready Scoring Pipeline (Single Customer)

**Steps implemented:**

1. Load persisted objects: model.h5, label_encoder_gender.pkl, one_hot_encoder_geo.pkl, scaler.pkl.
2. Assemble input as a DataFrame with the exact training feature names and column order(scaler.feature_names_in_).
3. Encode Gender (LabelEncoder) and one-hot Geography using the already-fitted encoders.
4. Concatenate encoded columns to numeric features; do not refit encoders/scaler.
5. Scale with the fitted StandardScaler.
6. Predict with ANN to obtain p = P(Exited=1 | x).
7. Decide using tuned threshold .

# 11. Recommendations & Next Steps

## 11.1 Improve Recall with Class Imbalance Handling

With churners making up only ~20% of the data, the model naturally becomes biased towards predicting "non-churn" to maximise accuracy. This leads to many missed churners (false negatives) and low recall.

Techniques like oversampling (e.g., SMOTE), undersampling, or class weighting give the model more exposure to the minority class during training, encouraging it to assign higher scores to churners and improving recall.

## 11.2 Hyperparameter Search

Default hyperparameters (layer sizes, learning rates, dropout rates, L2 penalties) are rarely optimal for a given dataset. A systematic search using tools like GridSearchCV, etc explores many combinations efficiently, often finding settings that improve convergence speed and metrics.

## 11.3 Add Batch Normalization

Batch Normalization works by adjusting the inputs to each layer so they're on a similar scale( roughly zero mean and unit variance) during training.

Why this helps:

- **Reduces "internal covariate shift"**: Every time earlier layers update their weights, the distribution of values going into the next layer can change (mean shifts, variance changes). Inputs are normalized to roughly the same mean and variance each time, so later layers see a more stable input distribution.
- **Acts like a light form of regularization**: BN adds small fluctuations because the mean and variance are computed per batch during training. This creates slight randomness in the normalized values (since batches differ), which can help prevent overfitting.
- **Speeds up convergence**: Keeps activations in a stable range so the model reaches good performance in fewer epochs.

## 11.4 Compare Against Tree Ensembles

Tree-based models like XGBoost, LightGBM, and Random Forests handle tabular, imbalanced datasets very well, often outperforming neural networks.
Comparing performance helps determine if the ANN is truly the best fit, or if a tree ensemble offers better recall, AUC, etc

## 11.9 K Fold CV

A single train/validation/test split can produce misleading results if the split isn't representative of the overall dataset. K-fold CV rotates through multiple splits, training and evaluating the model on different subsets of the data.