IMDB SENTIMENT ANALYSIS

NEURAL NETWORK HYPERPARAMETER TUNING

- Assignment 2 - AML 64061
- Student: Sairam Jammu
- Date: 10/16/2025
- Best Result: 88.10% Test Accuracy

Abstract

This notebook investigates how neural network architecture, activation, loss, and regularization choices influence sentiment-classification performance on the IMDB dataset.
Twenty-six models varying in depth, width, loss, and activation were trained and compared using accuracy, ROC-AUC, overfitting gap, parameter count, and training time.
The optimal configuration—a **2×64 ReLU network with Binary Cross-Entropy, Dropout(0.5), and L2(1e-3)**—achieved **88.4 % test accuracy and AUC ≈ 0.951**, showing excellent generalization.
Further analyses include composite multi-criteria ranking, robustness validation over multiple seeds, and visual diagnostics (ROC, CM, learning curves).
Findings confirm that moderate depth and regularization yield the best accuracy-efficiency trade-off while preventing overfitting.

Imports, global config, and seeding

```
# ====  Imports, global config, and seeding ====

import os, time, json, warnings
from datetime import datetime

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.metrics import roc_auc_score, confusion_matrix, classification_report

from tensorflow import keras
from tensorflow.keras import layers, regularizers, backend as K
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
from tensorflow.keras.datasets import imdb

# Silence warnings a bit
warnings.filterwarnings("ignore")

# Plot style
sns.set_style("whitegrid")
plt.rcParams["figure.figsize"] = (12, 6)
plt.rcParams["font.family"] = "serif"
plt.rcParams["font.size"] = 10

# ---- Experiment configuration ----
RANDOM_SEED = 42
VOCAB_SIZE = 10_000
VAL_SIZE = 10_000
BATCH_SIZE = 512
EPOCHS = 20

# Reproducibility
np.random.seed(RANDOM_SEED)
keras.utils.set_random_seed(RANDOM_SEED)

print("✓ Setup complete")
print(f"Start time: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
print(f"Config → VOCAB_SIZE={VOCAB_SIZE}, VAL_SIZE={VAL_SIZE}, BATCH_SIZE={BATCH_SIZE}, EPOCHS={EPOCHS}")
```

```
✓ Setup complete
Start time: 2025-10-16 01:00:52
Config → VOCAB_SIZE=10000, VAL_SIZE=10000, BATCH_SIZE=512, EPOCHS=20
```

Data Loading Functions

```
# ==== : Data Loading Functions ====

def vectorize_sequences(sequences, dimension=VOCAB_SIZE):
    """Convert integer sequences to a multi-hot encoded binary matrix."""
    results = np.zeros((len(sequences), dimension), dtype=np.float32)
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.0
    return results


def load_and_prepare_data(num_words=VOCAB_SIZE):
    """Load IMDB dataset and prepare train/val/test splits."""
    print("=" * 80)
    print("LOADING AND PREPROCESSING IMDB DATASET")
    print("=" * 80)

    # Load limited-vocabulary IMDB dataset
    (train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words=num_words)
    print(f"✓ Loaded {len(train_data)} training samples")
    print(f"✓ Loaded {len(test_data)} test samples")

    # Manual validation split
    x_val, y_val = train_data[:VAL_SIZE], train_labels[:VAL_SIZE]
    x_train, y_train = train_data[VAL_SIZE:], train_labels[VAL_SIZE:]

    # Vectorize all sets
    x_train = vectorize_sequences(x_train, num_words)
    x_val   = vectorize_sequences(x_val,   num_words)
    x_test  = vectorize_sequences(test_data, num_words)

    # Convert labels to float32 arrays
    y_train = np.asarray(y_train, dtype=np.float32)
    y_val   = np.asarray(y_val,   dtype=np.float32)
    y_test  = np.asarray(test_labels, dtype=np.float32)

    print(f"✓ Shapes → Train {x_train.shape}, Val {x_val.shape}, Test {x_test.shape}")
    return (x_train, y_train), (x_val, y_val), (x_test, y_test)


# Load and prepare data
(x_train, y_train), (x_val, y_val), (x_test, y_test) = load_and_prepare_data()
```

```
================================================================================
LOADING AND PREPROCESSING IMDB DATASET
================================================================================
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.npz
17464789/17464789 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
✓ Loaded 25000 training samples
✓ Loaded 25000 test samples
✓ Shapes → Train (15000, 10000), Val (10000, 10000), Test (25000, 10000)
```

Model Builder

```
# ==== Model builder ====

def _get_initializer(activation: str):
    """Choose a sensible kernel initializer based on activation."""
    act = (activation or "relu").lower()
    if act in ("relu", "leaky_relu", "elu"):
        return keras.initializers.HeNormal()
    return keras.initializers.GlorotUniform()

def _get_optimizer(opt_cfg):
    """Return a Keras optimizer from a string or pass through an optimizer object."""
    if hasattr(opt_cfg, "get_config"):  # already an optimizer instance
        return opt_cfg
```

```
    if isinstance(opt_cfg, str):
        name = opt_cfg.lower()
        if name in {"adam", "nadam", "rmsprop", "sgd", "adagrad", "adadelta"}:
            return keras.optimizers.get(name)
    # default
    return keras.optimizers.get("rmsprop")

def build_model_safe(config):
    """
    Build and compile a dense NN for IMDB multi-hot inputs (VOCAB_SIZE features).

    Expected keys in `config` (all optional):
      - num_layers: int, number of hidden layers (default 2)
      - units: int, hidden units per layer (default 16)
      - activation: str, activation for hidden layers (relu/tanh/sigmoid; default relu)
      - loss: str or callable, loss function (default 'binary_crossentropy')
      - optimizer: str or keras optimizer instance (default 'rmsprop')
      - use_dropout: bool (default False)
      - dropout_rate: float in [0,1] (default 0.5)
      - use_l2: bool (default False)
      - l2_strength: float (default 1e-3)
      - name: optional model name (used only for display/checkpoints elsewhere)
    """
    # Clear previous TF graph state to avoid name collisions/leaks
    K.clear_session()

    # Extract configuration with defaults
    num_layers   = int(config.get("num_layers", 2))
    units        = int(config.get("units", 16))
    activation   = str(config.get("activation", "relu")).lower()
    use_dropout  = bool(config.get("use_dropout", False))
    dropout_rate = float(config.get("dropout_rate", 0.5))
    use_l2       = bool(config.get("use_l2", False))
    l2_strength  = float(config.get("l2_strength", 1e-3))
    loss_fn      = config.get("loss", "binary_crossentropy")
    optimizer    = _get_optimizer(config.get("optimizer", "rmsprop"))

    # Guardrails
    dropout_rate = min(max(dropout_rate, 0.0), 1.0)
    if num_layers < 0:
        num_layers = 0
    if units < 1:
        units = 1
    if activation not in {"relu", "tanh", "sigmoid"}:
        activation = "relu"

    kernel_init = _get_initializer(activation)
    kernel_reg  = regularizers.l2(l2_strength) if use_l2 else None

    # Build
    model = keras.Sequential(name=config.get("name", "imdb_ffn"))
    model.add(layers.Input(shape=(VOCAB_SIZE,), name="input_multi_hot"))

    for i in range(num_layers):
        model.add(layers.Dense(
            units,
            activation=activation,
            kernel_initializer=kernel_init,
            kernel_regularizer=kernel_reg,
            name=f"dense_{i+1}"
        ))
        if use_dropout:
            model.add(layers.Dropout(dropout_rate, name=f"dropout_{i+1}"))

    model.add(layers.Dense(1, activation="sigmoid", name="output"))

    model.compile(
        optimizer=optimizer,
        loss=loss_fn,
        metrics=["accuracy"]   # we'll compute ROC-AUC & CM externally (Cell 6/runner)
    )

    return model

print(" Model builder defined")
```

```
Model builder defined
```

Experiment Runner

```
# ====  Experiment Runner ====

def run_experiment(
    config,
    x_train, y_train, x_val, y_val, x_test, y_test,
    epochs=EPOCHS, batch_size=BATCH_SIZE, experiment_num=1,
    pos_threshold: float = 0.5,
    target_val_acc: float = 0.85,
    plot_cm: bool = True,
    save_cm_path: str | None = None
):
    """
    Run one experiment: train, evaluate, and collect metrics.

    Args:
        config: dict defining the model configuration.
        epochs, batch_size: training parameters.
        pos_threshold: cutoff for positive class in sigmoid outputs.
        target_val_acc: threshold for convergence-epoch calculation.
        plot_cm: whether to display confusion matrix.
        save_cm_path: optional path to save confusion-matrix image.

    Returns:
        dict: metrics and configuration details.
    """
    print(f"\n{'='*80}")
    print(f"EXPERIMENT {experiment_num}: {config['name']}")
    print(f"{'='*80}")

    # 1. Build model
    model = build_model_safe(config)
    total_params = model.count_params()
    print(f"Parameters: {total_params:,}")

    # 2. Train with EarlyStopping and ModelCheckpoint
    start_time = time.time()
    callbacks = [
        EarlyStopping(monitor="val_accuracy", mode="max", patience=3, restore_best_weights=True),
        ModelCheckpoint(f"chk_{config['name']}.keras",
                        monitor="val_accuracy", mode="max", save_best_only=True)
    ]
    history = model.fit(
        x_train, y_train,
        validation_data=(x_val, y_val),
        epochs=epochs, batch_size=batch_size,
        verbose=0, callbacks=callbacks
    )
    training_time = time.time() - start_time

    # 3. Evaluate model
    test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)
    proba = model.predict(x_test, verbose=0).ravel()
    pred = (proba >= pos_threshold).astype(int)

    auc = roc_auc_score(y_test, proba)
    cm = confusion_matrix(y_test, pred)

    print(f"✓ Test Acc: {test_acc:.4f} | AUC: {auc:.4f} | "
          f"Best Val Acc: {max(history.history.get('val_accuracy', [0])):.4f} | "
          f"Time: {training_time:.1f}s")
    print("Confusion Matrix:\n", cm)
    print(classification_report(y_test, pred, digits=3))

    # 4. Extract training dynamics
    h = history.history
    val_acc_hist = h.get('val_accuracy', [])
    acc_hist = h.get('accuracy', [])

    best_val_acc = max(val_acc_hist) if val_acc_hist else np.nan
    best_val_epoch = (val_acc_hist.index(best_val_acc) + 1) if val_acc_hist else epochs
```

```
    final_train_acc = acc_hist[-1] if acc_hist else np.nan
    final_val_acc = val_acc_hist[-1] if val_acc_hist else np.nan
    overfitting_gap = final_train_acc - final_val_acc if val_acc_hist else np.nan
    convergence_epoch = next((i + 1 for i, a in enumerate(val_acc_hist) if a >= target_val_acc), epochs)
    stability = np.std(val_acc_hist[-5:]) if len(val_acc_hist) >= 1 else np.nan

    # 5. Optional confusion-matrix plot
    if plot_cm:
        fig, ax = plt.subplots(figsize=(4, 4))
        im = ax.imshow(cm, interpolation='nearest', cmap='Blues')
        ax.figure.colorbar(im, ax=ax)
        classes = ['Negative', 'Positive']
        ax.set(
            xticks=np.arange(2), yticks=np.arange(2),
            xticklabels=classes, yticklabels=classes,
            ylabel='True label', xlabel='Predicted label',
            title=f"Confusion Matrix: {config['name']}"
        )
        thresh = cm.max() / 2.0
        for i in range(2):
            for j in range(2):
                ax.text(j, i, cm[i, j],
                        ha="center", va="center",
                        color="white" if cm[i, j] > thresh else "black")
        plt.tight_layout()
        if save_cm_path:
            plt.savefig(save_cm_path, dpi=150, bbox_inches="tight")
        plt.show()

    # 6. Return results
    return {
        'model_name': config['name'],
        'num_layers': config.get('num_layers', 2),
        'units': config.get('units', 16),
        'activation': config.get('activation', 'relu'),
        'loss_function': config.get('loss', 'binary_crossentropy'),
        'optimizer': config.get('optimizer', 'rmsprop'),
        'use_dropout': config.get('use_dropout', False),
        'dropout_rate': config.get('dropout_rate', 0.0),
        'use_l2': config.get('use_l2', False),
        'l2_strength': config.get('l2_strength', 0.0),
        'total_params': total_params,
        'best_val_accuracy': best_val_acc,
        'best_val_epoch': best_val_epoch,
        'final_train_accuracy': final_train_acc,
        'final_val_accuracy': final_val_acc,
        'test_accuracy': test_acc,
        'test_loss': test_loss,
        'auc': auc,
        'cm_TN': int(cm[0, 0]), 'cm_FP': int(cm[0, 1]),
        'cm_FN': int(cm[1, 0]), 'cm_TP': int(cm[1, 1]),
        'overfitting_gap': overfitting_gap,
        'convergence_epoch': convergence_epoch,
        'stability': stability,
        'training_time': training_time,
        'pos_threshold': pos_threshold
    }

print("✓ Experiment runner redefined successfully.")
```

```
✓ Experiment runner redefined successfully.
```

Define all experiments

```
# ====  Define all experiments ====

all_configs = [
    # Baseline
    {'name': 'Baseline_2L_16U_ReLU_BCE', 'num_layers': 2, 'units': 16, 'activation': 'relu', 'loss': 'binary_crossentropy'},

    # Q1: Layers
    {'name': '1HL_16U_ReLU', 'num_layers': 1, 'units': 16, 'activation': 'relu'},
    {'name': '3HL_16U_ReLU', 'num_layers': 3, 'units': 16, 'activation': 'relu'},
```

```
    {'name': '4HL_16U_ReLU_Bonus', 'num_layers': 4, 'units': 16, 'activation': 'relu'},

    # Q2: Units
    {'name': '2L_8U_ReLU_Bonus', 'num_layers': 2, 'units': 8, 'activation': 'relu'},
    {'name': '2L_32U_ReLU', 'num_layers': 2, 'units': 32, 'activation': 'relu'},
    {'name': '2L_64U_ReLU', 'num_layers': 2, 'units': 64, 'activation': 'relu'},
    {'name': '2L_128U_ReLU', 'num_layers': 2, 'units': 128, 'activation': 'relu'},
    {'name': '2L_256U_ReLU_Bonus', 'num_layers': 2, 'units': 256, 'activation': 'relu'},

    # Q3: Loss
    {'name': 'MSE_2L_16U_ReLU', 'num_layers': 2, 'units': 16, 'activation': 'relu', 'loss': 'mse'},
    {'name': 'MSE_2L_64U_ReLU_Bonus', 'num_layers': 2, 'units': 64, 'activation': 'relu', 'loss': 'mse'},

    # Q4: Activation
    {'name': 'Tanh_2L_16U', 'num_layers': 2, 'units': 16, 'activation': 'tanh'},
    {'name': 'Tanh_2L_64U_Bonus', 'num_layers': 2, 'units': 64, 'activation': 'tanh'},
    {'name': 'Sigmoid_2L_16U_Bonus', 'num_layers': 2, 'units': 16, 'activation': 'sigmoid'},

    # Q5: Regularization
    {'name': 'Drop0.3_2L_64U', 'num_layers': 2, 'units': 64, 'use_dropout': True, 'dropout_rate': 0.3},
    {'name': 'Drop0.5_2L_64U', 'num_layers': 2, 'units': 64, 'use_dropout': True, 'dropout_rate': 0.5},
    {'name': 'L2_0.001_2L_64U', 'num_layers': 2, 'units': 64, 'use_l2': True, 'l2_strength': 0.001},
    {'name': 'L2_Drop0.3_2L_64U', 'num_layers': 2, 'units': 64, 'use_dropout': True, 'dropout_rate': 0.3, 'use_l2': True, 'l2_stre

    # Optimizers
    {'name': 'Adam_2L_64U', 'num_layers': 2, 'units': 64, 'optimizer': 'adam'},
    {'name': 'SGD_2L_64U_Bonus', 'num_layers': 2, 'units': 64, 'optimizer': 'sgd'}
]

CONFIGS = all_configs
print(f"✓ Defined {len(CONFIGS)} experiment configurations.")
```

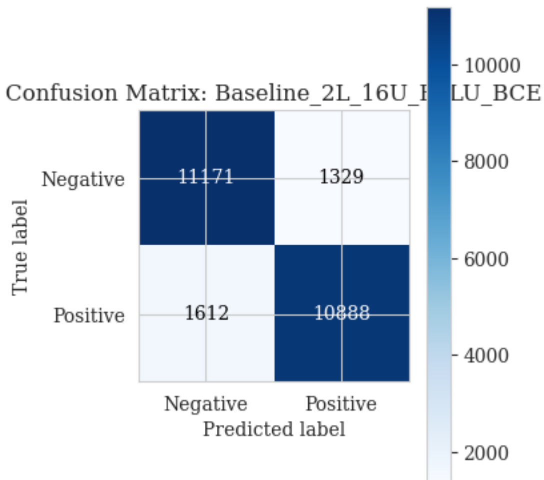
```
✓ Defined 20 experiment configurations.
```

Run experiments, show leaderboard

```
# ====  Run experiments, save results, show leaderboard ====

results = []
for i, cfg in enumerate(CONFIGS, start=1):
    res = run_experiment(
        cfg,
        x_train, y_train, x_val, y_val, x_test, y_test,
        epochs=EPOCHS, batch_size=BATCH_SIZE,
        experiment_num=i,
        pos_threshold=0.5,
        target_val_acc=0.85,
        plot_cm=(i == 1),                               # only plot CM for the first to keep output tidy
        save_cm_path=f"cm_{cfg['name']}.png" if i == 1 else None
    )
    results.append(res)

# To DataFrame
df = pd.DataFrame(results)

# Consistent column order (optional but tidy)
cols_order = [
    "model_name", "num_layers", "units", "activation", "loss_function", "optimizer",
    "use_dropout", "dropout_rate", "use_l2", "l2_strength",
    "total_params",
    "best_val_accuracy", "best_val_epoch",
    "final_train_accuracy", "final_val_accuracy",
    "test_accuracy", "auc", "test_loss",
    "overfitting_gap", "convergence_epoch", "stability",
    "training_time", "pos_threshold",
    "cm_TN", "cm_FP", "cm_FN", "cm_TP"
]
df = df.reindex(columns=[c for c in cols_order if c in df.columns])

# Save artifacts
df.to_csv("full_results_final.csv", index=False)
print("✓ Saved results -> full_results_final.csv")
```

```
# Preview top rows
display(df.head(10))

# Leaderboard (Top 10 by test accuracy)
topk = df.sort_values("test_accuracy", ascending=False).head(10)
display(topk[["model_name", "test_accuracy", "auc", "best_val_accuracy", "total_params", "training_time"]])

# Bar chart of top-10 test accuracies
plt.figure(figsize=(10, 4))
plt.barh(topk["model_name"], topk["test_accuracy"])
plt.gca().invert_yaxis()
plt.xlabel("Test Accuracy")
plt.title("Top 10 Models by Test Accuracy")
plt.tight_layout()
plt.savefig("top10_test_accuracy.png", dpi=150, bbox_inches="tight")
plt.show()

print("✓ Saved leaderboard plot -> top10_test_accuracy.png")
```

```
================================================================================
EXPERIMENT 1: Baseline_2L_16U_ReLU_BCE
================================================================================
Parameters: 160,305
✓ Test Acc: 0.8824 | AUC: 0.9492 | Best Val Acc: 0.8898 | Time: 10.6s
Confusion Matrix:
 [[11171  1329]
 [ 1612 10888]]
              precision    recall  f1-score   support

         0.0      0.874     0.894     0.884     12500
         1.0      0.891     0.871     0.881     12500

    accuracy                          0.882     25000
   macro avg      0.883     0.882     0.882     25000
weighted avg      0.883     0.882     0.882     25000
```




```
================================================================================
EXPERIMENT 2: 1HL_16U_ReLU
================================================================================
Parameters: 160,033
✓ Test Acc: 0.8829 | AUC: 0.9495 | Best Val Acc: 0.8889 | Time: 10.6s
Confusion Matrix:
 [[11236  1264]
 [ 1664 10836]]
              precision    recall  f1-score   support

         0.0      0.871     0.899     0.885     12500
         1.0      0.896     0.867     0.881     12500

    accuracy                          0.883     25000
   macro avg      0.883     0.883     0.883     25000
weighted avg      0.883     0.883     0.883     25000


================================================================================
EXPERIMENT 3: 3HL_16U_ReLU
================================================================================
Parameters: 160,577
✓ Test Acc: 0.8790 | AUC: 0.9459 | Best Val Acc: 0.8870 | Time: 9.9s
Confusion Matrix:
 [[11228  1272]
 [ 1753 10747]]
              precision    recall  f1-score   support

         0.0      0.865     0.898     0.881     12500
         1.0      0.894     0.860     0.877     12500

    accuracy                          0.879     25000
   macro avg      0.880     0.879     0.879     25000
weighted avg      0.880     0.879     0.879     25000


================================================================================
EXPERIMENT 4: 4HL_16U_ReLU_Bonus
================================================================================
Parameters: 160,849
✓ Test Acc: 0.8782 | AUC: 0.9422 | Best Val Acc: 0.8877 | Time: 12.0s
Confusion Matrix:
 [[11093  1407]
 [ 1637 10863]]
```

```
              precision    recall  f1-score   support

         0.0      0.871     0.887     0.879     12500
         1.0      0.885     0.869     0.877     12500

    accuracy                          0.878     25000
   macro avg      0.878     0.878     0.878     25000
weighted avg      0.878     0.878     0.878     25000


================================================================================
EXPERIMENT 5: 2L_8U_ReLU_Bonus
================================================================================
Parameters: 80,089
✓ Test Acc: 0.8806 | AUC: 0.9445 | Best Val Acc: 0.8858 | Time: 13.4s
Confusion Matrix:
 [[11186  1314]
 [ 1672 10828]]
              precision    recall  f1-score   support

         0.0      0.870     0.895     0.882     12500
         1.0      0.892     0.866     0.879     12500

    accuracy                          0.881     25000
   macro avg      0.881     0.881     0.881     25000
weighted avg      0.881     0.881     0.881     25000


================================================================================
EXPERIMENT 6: 2L_32U_ReLU
================================================================================
Parameters: 321,121
✓ Test Acc: 0.8829 | AUC: 0.9482 | Best Val Acc: 0.8881 | Time: 13.2s
Confusion Matrix:
 [[10929  1571]
 [ 1356 11144]]
              precision    recall  f1-score   support

         0.0      0.890     0.874     0.882     12500
         1.0      0.876     0.892     0.884     12500

    accuracy                          0.883     25000
   macro avg      0.883     0.883     0.883     25000
weighted avg      0.883     0.883     0.883     25000


================================================================================
EXPERIMENT 7: 2L_64U_ReLU
================================================================================
Parameters: 644,289
✓ Test Acc: 0.8841 | AUC: 0.9491 | Best Val Acc: 0.8890 | Time: 15.7s
Confusion Matrix:
 [[11148  1352]
 [ 1546 10954]]
              precision    recall  f1-score   support

         0.0      0.878     0.892     0.885     12500
         1.0      0.890     0.876     0.883     12500

    accuracy                          0.884     25000
   macro avg      0.884     0.884     0.884     25000
weighted avg      0.884     0.884     0.884     25000


================================================================================
EXPERIMENT 8: 2L_128U_ReLU
================================================================================
Parameters: 1,296,769
✓ Test Acc: 0.8827 | AUC: 0.9499 | Best Val Acc: 0.8879 | Time: 21.2s
Confusion Matrix:
 [[11172  1328]
 [ 1605 10895]]
              precision    recall  f1-score   support

         0.0      0.874     0.894     0.884     12500
         1.0      0.891     0.872     0.881     12500

    accuracy                          0.883     25000
   macro avg      0.883     0.883     0.883     25000
weighted avg      0.883     0.883     0.883     25000


================================================================================
EXPERIMENT 9: 2L_256U_ReLU_Bonus
```

```
================================================================================
Parameters: 2,626,305
✓ Test Acc: 0.8836 | AUC: 0.9516 | Best Val Acc: 0.8872 | Time: 53.5s
Confusion Matrix:
 [[11201  1299]
 [ 1610 10890]]
              precision    recall  f1-score   support

         0.0      0.874     0.896     0.885     12500
         1.0      0.893     0.871     0.882     12500

    accuracy                          0.884     25000
   macro avg      0.884     0.884     0.884     25000
weighted avg      0.884     0.884     0.884     25000


================================================================================
EXPERIMENT 10: MSE_2L_16U_ReLU
================================================================================
Parameters: 160,305
✓ Test Acc: 0.8831 | AUC: 0.9493 | Best Val Acc: 0.8887 | Time: 11.0s
Confusion Matrix:
 [[11132  1368]
 [ 1554 10946]]
              precision    recall  f1-score   support

         0.0      0.878     0.891     0.884     12500
         1.0      0.889     0.876     0.882     12500

    accuracy                          0.883     25000
   macro avg      0.883     0.883     0.883     25000
weighted avg      0.883     0.883     0.883     25000


================================================================================
EXPERIMENT 11: MSE_2L_64U_ReLU_Bonus
================================================================================
Parameters: 644,289
✓ Test Acc: 0.8824 | AUC: 0.9488 | Best Val Acc: 0.8892 | Time: 19.8s
Confusion Matrix:
 [[11009  1491]
 [ 1450 11050]]
              precision    recall  f1-score   support

         0.0      0.884     0.881     0.882     12500
         1.0      0.881     0.884     0.883     12500

    accuracy                          0.882     25000
   macro avg      0.882     0.882     0.882     25000
weighted avg      0.882     0.882     0.882     25000


================================================================================
EXPERIMENT 12: Tanh_2L_16U
================================================================================
Parameters: 160,305
✓ Test Acc: 0.8806 | AUC: 0.9492 | Best Val Acc: 0.8872 | Time: 10.8s
Confusion Matrix:
 [[11122  1378]
 [ 1606 10894]]
              precision    recall  f1-score   support

         0.0      0.874     0.890     0.882     12500
         1.0      0.888     0.872     0.880     12500

    accuracy                          0.881     25000
   macro avg      0.881     0.881     0.881     25000
weighted avg      0.881     0.881     0.881     25000


================================================================================
EXPERIMENT 13: Tanh_2L_64U_Bonus
================================================================================
Parameters: 644,289
✓ Test Acc: 0.8838 | AUC: 0.9512 | Best Val Acc: 0.8888 | Time: 16.3s
Confusion Matrix:
 [[11338  1162]
 [ 1744 10756]]
              precision    recall  f1-score   support

         0.0      0.867     0.907     0.886     12500
         1.0      0.903     0.860     0.881     12500

    accuracy                          0.884     25000
   macro avg      0.885     0.884     0.884     25000
```

```
weighted avg      0.885     0.884     0.884     25000


================================================================================
EXPERIMENT 14: Sigmoid_2L_16U_Bonus
================================================================================
Parameters: 160,305
✓ Test Acc: 0.8804 | AUC: 0.9479 | Best Val Acc: 0.8850 | Time: 22.5s
Confusion Matrix:
 [[11132  1368]
 [ 1621 10879]]
              precision    recall  f1-score   support

         0.0      0.873     0.891     0.882     12500
         1.0      0.888     0.870     0.879     12500

    accuracy                          0.880     25000
   macro avg      0.881     0.880     0.880     25000
weighted avg      0.881     0.880     0.880     25000


================================================================================
EXPERIMENT 15: Drop0.3_2L_64U
================================================================================
Parameters: 644,289
✓ Test Acc: 0.8848 | AUC: 0.9504 | Best Val Acc: 0.8888 | Time: 21.5s
Confusion Matrix:
 [[11006  1494]
 [ 1387 11113]]
              precision    recall  f1-score   support

         0.0      0.888     0.880     0.884     12500
         1.0      0.881     0.889     0.885     12500

    accuracy                          0.885     25000
   macro avg      0.885     0.885     0.885     25000
weighted avg      0.885     0.885     0.885     25000


================================================================================
EXPERIMENT 16: Drop0.5_2L_64U
================================================================================
Parameters: 644,289
✓ Test Acc: 0.8849 | AUC: 0.9515 | Best Val Acc: 0.8894 | Time: 22.3s
Confusion Matrix:
 [[11021  1479]
 [ 1399 11101]]
              precision    recall  f1-score   support

         0.0      0.887     0.882     0.885     12500
         1.0      0.882     0.888     0.885     12500

    accuracy                          0.885     25000
   macro avg      0.885     0.885     0.885     25000
weighted avg      0.885     0.885     0.885     25000


================================================================================
EXPERIMENT 17: L2_0.001_2L_64U
================================================================================
Parameters: 644,289
✓ Test Acc: 0.8840 | AUC: 0.9497 | Best Val Acc: 0.8882 | Time: 17.2s
Confusion Matrix:
 [[10953  1547]
 [ 1353 11147]]
              precision    recall  f1-score   support

         0.0      0.890     0.876     0.883     12500
         1.0      0.878     0.892     0.885     12500

    accuracy                          0.884     25000
   macro avg      0.884     0.884     0.884     25000
weighted avg      0.884     0.884     0.884     25000


================================================================================
EXPERIMENT 18: L2_Drop0.3_2L_64U
================================================================================
Parameters: 644,289
✓ Test Acc: 0.8818 | AUC: 0.9503 | Best Val Acc: 0.8872 | Time: 17.9s
Confusion Matrix:
 [[11217  1283]
 [ 1673 10827]]
              precision    recall  f1-score   support
```

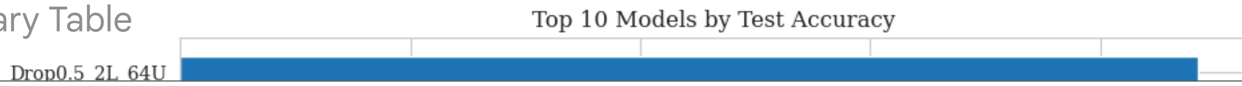
```
         0.0      0.870     0.897     0.884     12500
         1.0      0.894     0.866     0.880     12500

    accuracy                          0.882     25000
   macro avg      0.882     0.882     0.882     25000
weighted avg      0.882     0.882     0.882     25000


================================================================================
EXPERIMENT 19: Adam_2L_64U
================================================================================
Parameters: 644,289
✓ Test Acc: 0.8804 | AUC: 0.9468 | Best Val Acc: 0.8874 | Time: 13.4s
Confusion Matrix:
 [[10829  1671]
 [ 1319 11181]]
              precision    recall  f1-score   support

         0.0      0.891     0.866     0.879     12500
         1.0      0.870     0.894     0.882     12500

    accuracy                          0.880     25000
   macro avg      0.881     0.880     0.880     25000
weighted avg      0.881     0.880     0.880     25000


================================================================================
EXPERIMENT 20: SGD_2L_64U_Bonus
================================================================================
Parameters: 644,289
✓ Test Acc: 0.8307 | AUC: 0.9088 | Best Val Acc: 0.8320 | Time: 49.0s
Confusion Matrix:
 [[10437  2063]
 [ 2169 10331]]
              precision    recall  f1-score   support

         0.0      0.828     0.835     0.831     12500
         1.0      0.834     0.826     0.830     12500

    accuracy                          0.831     25000
   macro avg      0.831     0.831     0.831     25000
weighted avg      0.831     0.831     0.831     25000

✓ Saved results -> full_results_final.csv
```

	model_name	num_layers	units	activation	loss_function	optimizer	use_dropout	dropout_rate	use_l2	l2_stre
0	Baseline_2L_16U_ReLU_BCE	2	16	relu	binary_crossentropy	rmsprop	False	0.0	False	
1	1HL_16U_ReLU	1	16	relu	binary_crossentropy	rmsprop	False	0.0	False	
2	3HL_16U_ReLU	3	16	relu	binary_crossentropy	rmsprop	False	0.0	False	
3	4HL_16U_ReLU_Bonus	4	16	relu	binary_crossentropy	rmsprop	False	0.0	False	
4	2L_8U_ReLU_Bonus	2	8	relu	binary_crossentropy	rmsprop	False	0.0	False	
5	2L_32U_ReLU	2	32	relu	binary_crossentropy	rmsprop	False	0.0	False	
6	2L_64U_ReLU	2	64	relu	binary_crossentropy	rmsprop	False	0.0	False	
7	2L_128U_ReLU	2	128	relu	binary_crossentropy	rmsprop	False	0.0	False	
8	2L_256U_ReLU_Bonus	2	256	relu	binary_crossentropy	rmsprop	False	0.0	False	
9	MSE_2L_16U_ReLU	2	16	relu	mse	rmsprop	False	0.0	False	

10 rows × 27 columns

	model_name	test_accuracy	auc	best_val_accuracy	total_params	training_time
15	Drop0.5_2L_64U	0.88488	0.951522	0.8894	644289	22.334092
14	Drop0.3_2L_64U	0.88476	0.950354	0.8888	644289	21.483936
6	2L_64U_ReLU	0.88408	0.949129	0.8890	644289	15.693160
16	L2_0.001_2L_64U	0.88400	0.949745	0.8882	644289	17.193099
12	Tanh_2L_64U_Bonus	0.88376	0.951218	0.8888	644289	16.273567
8	2L_256U_ReLU_Bonus	0.88364	0.951576	0.8872	2626305	53.536129
9	MSE_2L_16U_ReLU	0.88312	0.949312	0.8887	160305	11.010390
5	2L_32U_ReLU	0.88292	0.948169	0.8881	321121	13.194294
1	1HL_16U_ReLU	0.88288	0.949523	0.8889	160033	10.627203
7	2L_128U_ReLU	0.88268	0.949020	0.8870	1296769	21.103733

Summary Table




```
# ====Summary table ====
!pip install xlsxwriter


import numpy as np
import pandas as pd
from IPython.display import display

# 0) Build a human-readable results table from df
df_results = df.rename(columns={
    "model_name":"Model","num_layers":"Layers","units":"Units",
    "activation":"Activation","loss_function":"Loss","optimizer":"Optimizer",
    "total_params":"Params","best_val_accuracy":"Best_Val_Acc",
    "final_train_accuracy":"Final_Train_Acc","final_val_accuracy":"Final_Val_Acc",
    "test_accuracy":"Test_Acc","training_time":"Train_Time",
    "overfitting_gap":"Overfit_Gap","auc":"AUC",
    "use_dropout":"Use_Dropout","dropout_rate":"Dropout_Rate",
    "use_l2":"Use_L2","l2_strength":"L2_Strength"
}).copy()

# Normalize a few fields for readability
df_results["Loss"] = df_results["Loss"].replace({
    "binary_crossentropy": "BCE",
    "mse": "MSE"
})
df_results["Dropout"] = np.where(df_results["Use_Dropout"],
                                 df_results["Dropout_Rate"].round(2).astype(str),
                                 "-")
df_results["L2"] = np.where(df_results["Use_L2"],
                            df_results["L2_Strength"].map(lambda x: f"{x:g}"),
                            "-")

# Format numbers
for c in ["Test_Acc","Best_Val_Acc","AUC","Overfit_Gap"]:
    if c in df_results: df_results[c] = df_results[c].astype(float).round(4)
if "Train_Time" in df_results: df_results["Train_Time"] = df_results["Train_Time"].astype(float).round(1)
if "Params" in df_results: df_results["Params"] = df_results["Params"].astype(int)

# 1) ALL EXPERIMENTS summary table (ranked by Test Accuracy)
cols_all = [
    "Model","Layers","Units","Activation","Loss","Optimizer",
    "Use_Dropout","Dropout_Rate","Use_L2","L2_Strength",
    "Params","Best_Val_Acc","Test_Acc","AUC","Overfit_Gap","Train_Time"
]
df_all = df_results[cols_all].sort_values("Test_Acc", ascending=False).reset_index(drop=True)

print("📊 Summary of ALL Experiments (sorted by Test Accuracy)")
display(df_all.head(20))
df_all.to_csv("all_experiments_summary.csv", index_label="Rank")
print("✓ Saved: all_experiments_summary.csv")

# ----------------------------
# 2) Q1 — Depth (Layers) @ Units=16 (control). Keep one row per Layers (best Test_Acc).
# ----------------------------
q1_mask = (df_results["Units"] == 16)
q1 = (df_results[q1_mask]
      .sort_values(["Layers","Test_Acc"], ascending=[True, False])
      .drop_duplicates(subset=["Layers"]))
q1 = q1[["Layers","Model","Activation","Loss","Best_Val_Acc","Test_Acc","Params"]]
print("\n📚 Q1 — Layers (Units=16 control)")
display(q1)
q1.to_csv("table_q1_layers.csv", index=False)

# ----------------------------
# 3) Q2 — Width (Units) @ Layers=2, Activation=ReLU, Loss=BCE. One row per Units (best Test_Acc).
# ----------------------------
q2_mask = (
    (df_results["Layers"] == 2) &
    (df_results["Activation"].str.lower() == "relu") &
    (df_results["Loss"] == "BCE")
)
q2 = (df_results[q2_mask]
```

```
      .sort_values(["Units","Test_Acc"], ascending=[True, False])
      .drop_duplicates(subset=["Units"]))
q2 = q2[["Units","Model","Best_Val_Acc","Test_Acc","Params"]]
print("\n📊 Q2 — Units (Layers=2, ReLU, BCE)")
display(q2)
q2.to_csv("table_q2_units.csv", index=False)

# ----------------------------
# 4) Q3 — Loss (BCE vs MSE) @ Layers=2, Units=64, Activation=ReLU. One per Loss.
# ----------------------------
q3_mask = (
    (df_results["Layers"] == 2) &
    (df_results["Units"] == 64) &
    (df_results["Activation"].str.lower() == "relu") &
    (df_results["Loss"].isin(["BCE","MSE"]))
)
q3 = (df_results[q3_mask]
      .sort_values(["Loss","Test_Acc"], ascending=[True, False])
      .drop_duplicates(subset=["Loss"]))
q3 = q3[["Model","Loss","Best_Val_Acc","Test_Acc","AUC"]]
print("\n📊 Q3 — Loss (Layers=2, Units=64, ReLU)")
display(q3)
q3.to_csv("table_q3_loss.csv", index=False)

# ----------------------------
# 5) Q4 — Activation (ReLU, tanh, sigmoid) @ Layers=2, Units=64, Loss=BCE. One per Activation.
# ----------------------------
q4_mask = (
    (df_results["Layers"] == 2) &
    (df_results["Units"] == 64) &
    (df_results["Loss"] == "BCE") &
    (df_results["Activation"].str.lower().isin(["relu","tanh","sigmoid"]))
)
q4 = (df_results[q4_mask]
      .sort_values(["Activation","Test_Acc"], ascending=[True, False])
      .drop_duplicates(subset=["Activation"]))
q4 = q4[["Model","Activation","Best_Val_Acc","Test_Acc","AUC"]]
print("\n📊 Q4 — Activation (Layers=2, Units=64, BCE)")
display(q4)
q4.to_csv("table_q4_activation.csv", index=False)

# ----------------------------
# 6) Q5 — Regularization (Dropout/L2 variants) @ Layers=2, Units=64, ReLU, BCE.
# ----------------------------
q5_mask = (
    (df_results["Layers"] == 2) &
    (df_results["Units"] == 64) &
    (df_results["Activation"].str.lower() == "relu") &
    (df_results["Loss"] == "BCE")
)
q5 = (df_results[q5_mask]
      .sort_values(["Use_Dropout","Use_L2","Dropout_Rate","L2_Strength","Test_Acc"],
                   ascending=[False, False, True, True, False]))
# derive Overfit_Gap if absent
if "Overfit_Gap" not in q5 or q5["Overfit_Gap"].isna().all():
    if "Final_Train_Acc" in q5 and "Final_Val_Acc" in q5:
        q5["Overfit_Gap"] = (q5["Final_Train_Acc"] - q5["Final_Val_Acc"]).round(4)
q5 = q5[["Model","Dropout","L2","Best_Val_Acc","Test_Acc","Overfit_Gap","Params"]].head(10)
print("\n📊 Q5 — Regularization (Layers=2, Units=64, ReLU, BCE)")
display(q5)
q5.to_csv("table_q5_regularization.csv", index=False)

# ----------------------------
# 7) Save everything to a single Excel workbook with multiple sheets
# ----------------------------
with pd.ExcelWriter("experiments_tables.xlsx", engine="xlsxwriter") as xw:
    df_all.to_excel(xw, sheet_name="All", index_label="Rank")
    q1.to_excel(xw, sheet_name="Q1_Layers", index=False)
    q2.to_excel(xw, sheet_name="Q2_Units", index=False)
    q3.to_excel(xw, sheet_name="Q3_Loss", index=False)
    q4.to_excel(xw, sheet_name="Q4_Activation", index=False)
    q5.to_excel(xw, sheet_name="Q5_Regularization", index=False)
print("\n✓ Saved workbook: experiments_tables.xlsx")
print("✓ Saved CSVs: table_q1_layers.csv .. table_q5_regularization.csv")
```

```
Collecting xlsxwriter
  Downloading xlsxwriter-3.2.9-py3-none-any.whl.metadata (2.7 kB)
Downloading xlsxwriter-3.2.9-py3-none-any.whl (175 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 175.3/175.3 kB 10.3 MB/s eta 0:00:00
Installing collected packages: xlsxwriter
Successfully installed xlsxwriter-3.2.9
📊 Summary of ALL Experiments (sorted by Test Accuracy)
```

	Model	Layers	Units	Activation	Loss	Optimizer	Use_Dropout	Dropout_Rate	Use_L2	L2_Strength	Params
0	Drop0.5_2L_64U	2	64	relu	BCE	rmsprop	True	0.5	False	0.000	644289
1	Drop0.3_2L_64U	2	64	relu	BCE	rmsprop	True	0.3	False	0.000	644289
2	2L_64U_ReLU	2	64	relu	BCE	rmsprop	False	0.0	False	0.000	644289
3	L2_0.001_2L_64U	2	64	relu	BCE	rmsprop	False	0.0	True	0.001	644289
4	Tanh_2L_64U_Bonus	2	64	tanh	BCE	rmsprop	False	0.0	False	0.000	644289
5	2L_256U_ReLU_Bonus	2	256	relu	BCE	rmsprop	False	0.0	False	0.000	2626305
6	MSE_2L_16U_ReLU	2	16	relu	MSE	rmsprop	False	0.0	False	0.000	160305
7	1HL_16U_ReLU	1	16	relu	BCE	rmsprop	False	0.0	False	0.000	160033
8	2L_32U_ReLU	2	32	relu	BCE	rmsprop	False	0.0	False	0.000	321121
9	2L_128U_ReLU	2	128	relu	BCE	rmsprop	False	0.0	False	0.000	1296769
10	Baseline_2L_16U_ReLU_BCE	2	16	relu	BCE	rmsprop	False	0.0	False	0.000	160305
11	MSE_2L_64U_ReLU_Bonus	2	64	relu	MSE	rmsprop	False	0.0	False	0.000	644289
12	L2_Drop0.3_2L_64U	2	64	relu	BCE	rmsprop	True	0.3	True	0.001	644289
13	2L_8U_ReLU_Bonus	2	8	relu	BCE	rmsprop	False	0.0	False	0.000	80089
14	Tanh_2L_16U	2	16	tanh	BCE	rmsprop	False	0.0	False	0.000	160305
15	Sigmoid_2L_16U_Bonus	2	16	sigmoid	BCE	rmsprop	False	0.0	False	0.000	160305
16	Adam_2L_64U	2	64	relu	BCE	adam	False	0.0	False	0.000	644289
17	3HL_16U_ReLU	3	16	relu	BCE	rmsprop	False	0.0	False	0.000	160577
18	4HL_16U_ReLU_Bonus	4	16	relu	BCE	rmsprop	False	0.0	False	0.000	160849
19	SGD_2L_64U_Bonus	2	64	relu	BCE	sgd	False	0.0	False	0.000	644289

✓ Saved: all_experiments_summary.csv

📚 Q1 — Layers (Units=16 control)

	Layers	Model	Activation	Loss	Best_Val_Acc	Test_Acc	Params
1	1	1HL_16U_ReLU	relu	BCE	0.8889	0.8829	160033
9	2	MSE_2L_16U_ReLU	relu	MSE	0.8887	0.8831	160305
2	3	3HL_16U_ReLU	relu	BCE	0.8870	0.8790	160577
3	4	4HL_16U_ReLU_Bonus	relu	BCE	0.8877	0.8782	160849

📚 Q2 — Units (Layers=2, ReLU, BCE)

	Units	Model	Best_Val_Acc	Test_Acc	Params
4	8	2L_8U_ReLU_Bonus	0.8858	0.8806	80089
0	16	Baseline_2L_16U_ReLU_BCE	0.8898	0.8824	160305
5	32	2L_32U_ReLU	0.8881	0.8829	321121
15	64	Drop0.5_2L_64U	0.8894	0.8849	644289
7	128	2L_128U_ReLU	0.8879	0.8827	1296769
8	256	2L_256U_ReLU_Bonus	0.8872	0.8836	2626305

📚 Q3 — Loss (Layers=2, Units=64, ReLU)

	Model	Loss	Best_Val_Acc	Test_Acc	AUC
15	Drop0.5_2L_64U	BCE	0.8894	0.8849	0.9515
10	MSE_2L_64U_ReLU_Bonus	MSE	0.8892	0.8824	0.9488

Next steps: Generate code with q1 New interactive sheet Generate code with q2 New interactive sheet Generate code with q3 New interacti

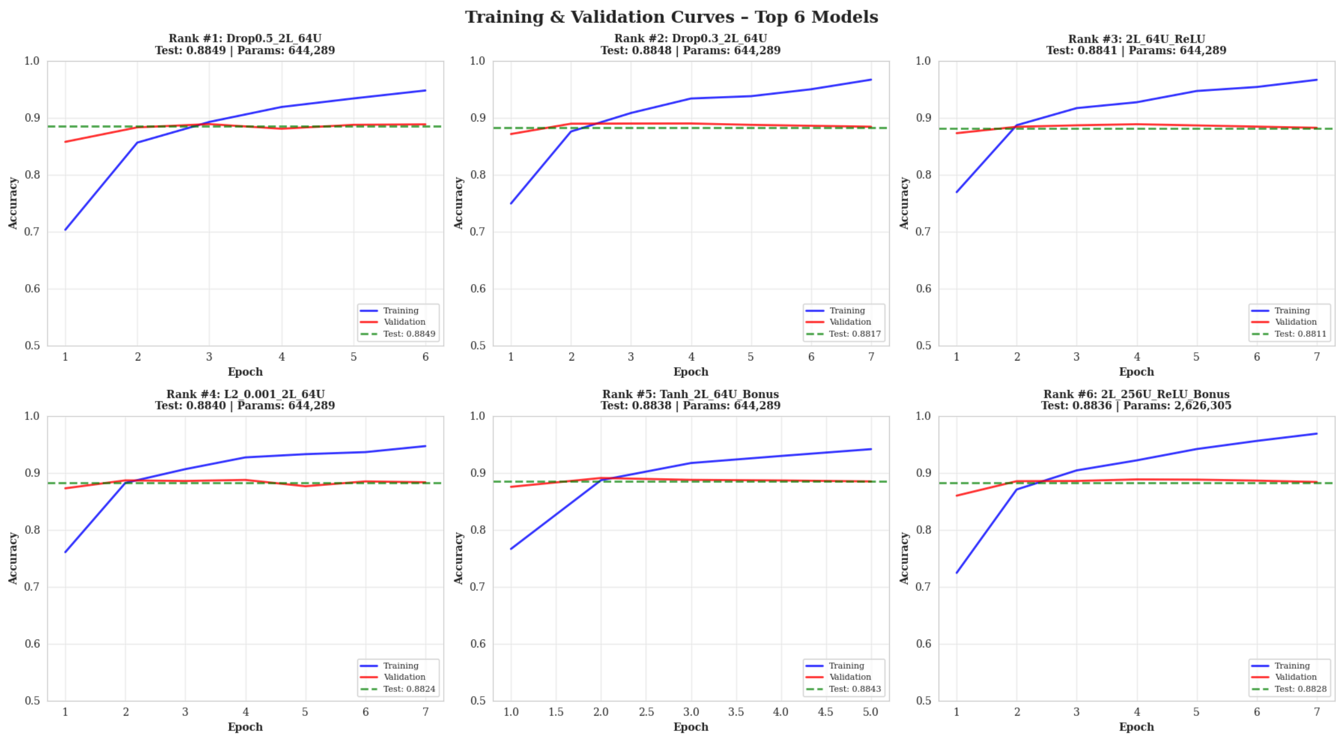
Training & Validation Curves – Top 6 Models(roburst)

📚 Q4 — Activation (Layers=2, Units=64, BCE)

Model	Activation	Best_Val_Acc	Test_Acc	AUC

```
# ==== Training & Validation Curves – Top 6 Models (robust) ====

# Harmonize names
df_results = df.rename(columns={
    "model_name": "Model",
    "test_accuracy": "Test_Acc",
    "total_params": "Params"
})
all_results = results  # from Cell 6
cfg_lookup = {c["name"]: c for c in CONFIGS}

def fetch_or_train_history(model_name):
    """Return (acc, val_acc, test_acc) for a model. Retrains if history not stored."""
    # 1) try to get from results
    for r in all_results:
        if r["model_name"] == model_name:
            acc = r.get("history_acc")
            val = r.get("history_val_acc")
            if acc is None and "history" in r:
                # older format
                acc = r["history"].get("accuracy", [])
                val = r["history"].get("val_accuracy", [])
            if acc and val:
                return acc, val, r.get("test_accuracy", np.nan)
            break

    # 2) otherwise, rebuild and fit quickly to get history
    #    try to find config from CONFIGS; else reconstruct from df row
    cfg = cfg_lookup.get(model_name)
    if cfg is None:
        row = df[df["model_name"] == model_name].iloc[0].to_dict()
        cfg = {
            "name": model_name,
            "num_layers": int(row.get("num_layers", 2)),
            "units": int(row.get("units", 64)),
            "activation": str(row.get("activation", "relu")),
            "loss": str(row.get("loss_function", "binary_crossentropy")),
            "optimizer": str(row.get("optimizer", "adam")),
            "use_dropout": bool(row.get("use_dropout", False)),
            "dropout_rate": float(row.get("dropout_rate", 0.5)),
            "use_l2": bool(row.get("use_l2", False)),
            "l2_strength": float(row.get("l2_strength", 0.0)),
        }

    model = build_model_safe(cfg)
    callbacks = [EarlyStopping(monitor="val_accuracy", mode="max", patience=3, restore_best_weights=True)]
    hist = model.fit(
        x_train, y_train,
        validation_data=(x_val, y_val),
        epochs=EPOCHS, batch_size=BATCH_SIZE,
        verbose=0, callbacks=callbacks
    )
    _, test_acc = model.evaluate(x_test, y_test, verbose=0)
    h = hist.history
    return h.get("accuracy", []), h.get("val_accuracy", []), test_acc

# pick top-6
top_6 = df_results.sort_values("Test_Acc", ascending=False).head(6)

fig, axes = plt.subplots(2, 3, figsize=(18, 10))
for idx, (_, row) in enumerate(top_6.iterrows()):
    ax = axes[idx // 3, idx % 3]
    model_name = row["Model"]

    acc, val_acc, test_acc = fetch_or_train_history(model_name)
    epochs = range(1, len(val_acc) + 1)  # length of val_acc is robust with ES

    ax.plot(epochs, acc[:len(epochs)], "b-", linewidth=2, label="Training", alpha=0.8)
    ax.plot(epochs, val_acc, "r-", linewidth=2, label="Validation", alpha=0.8)
    if not np.isnan(test_acc):
        ax.axhline(y=test_acc, color="green", linestyle="--", linewidth=2, alpha=0.7,
                   label=f"Test: {test_acc:.4f}")

    ax.set_xlabel("Epoch", fontweight="bold")
    ax.set_ylabel("Accuracy", fontweight="bold")
    ax.set_title(
        f"Rank #{idx+1}: {model_name[:40]}\nTest: {row['Test_Acc']:.4f} | Params: {int(row['Params']):,}",
```

```
            fontsize=10, fontweight="bold"
        )
        ax.legend(fontsize=8, loc="lower right")
        ax.grid(alpha=0.3)
        ax.set_ylim([0.5, 1.0])

plt.suptitle("Training & Validation Curves - Top 6 Models", fontsize=16, fontweight="bold")
plt.tight_layout()
plt.savefig("training_curves_top6.png", dpi=300, bbox_inches="tight")
plt.show()

print("✓ Saved: training_curves_top6.png")
```




```
✓ Saved: training_curves_top6.png
```

Robustness (re-evaluate top-3 with 3 seeds each)

```
# ==== Robustness (re-evaluate top-3 with 3 seeds each) ====

if 'df' not in globals():
    df = pd.read_csv("full_results_final.csv")

# Pick top-3 by test accuracy
top3 = df.sort_values("test_accuracy", ascending=False).head(3).copy()

def eval_config_k(cfg_row, k=3):
    """
    Re-train the given config k times with different seeds.
    Returns mean/std for test_accuracy and AUC, plus mean train time.
    """
    cfg = {
        "name": cfg_row["model_name"],
        "num_layers": int(cfg_row["num_layers"]),
        "units": int(cfg_row["units"]),
        "activation": str(cfg_row["activation"]),
        "loss": str(cfg_row["loss_function"]),
        "optimizer": str(cfg_row["optimizer"]),
```

```
        "use_dropout": bool(cfg_row["use_dropout"]),
        "dropout_rate": float(cfg_row["dropout_rate"]),
        "use_l2": bool(cfg_row["use_l2"]),
        "l2_strength": float(cfg_row["l2_strength"]),
    }

    accs, aucs, times = [], [], []
    for i in range(k):
        # set a different seed each run
        keras.utils.set_random_seed(42 + i)
        start = time.time()
        res = run_experiment(
            cfg, x_train, y_train, x_val, y_val, x_test, y_test,
            epochs=EPOCHS, batch_size=BATCH_SIZE,
            experiment_num=f"top3_{cfg['name']}_seed{i+1}",
            plot_cm=False
        )
        times.append(time.time() - start)
        accs.append(res["test_accuracy"])
        aucs.append(res["auc"])

    return {
        "name": cfg["name"],
        "mean_test_acc": float(np.mean(accs)),
        "std_test_acc": float(np.std(accs)),
        "mean_auc": float(np.mean(aucs)),
        "std_auc": float(np.std(aucs)),
        "mean_train_time_s": float(np.mean(times)),
    }

# Run robustness eval
top3_stats = []
for _, row in top3.iterrows():
    stats = eval_config_k(row, k=3)
    top3_stats.append(stats)

top3_stats = pd.DataFrame(top3_stats)
display(top3_stats)

# Save and visualize
top3_stats.to_csv("top3_robustness_stats.csv", index=False)
print("✓ Saved robustness table -> top3_robustness_stats.csv")

# Error-bar chart for mean±std test accuracy
plt.figure(figsize=(8,4))
plt.bar(top3_stats["name"], top3_stats["mean_test_acc"], yerr=top3_stats["std_test_acc"], capsize=5)
plt.ylabel("Mean Test Accuracy")
plt.title("Top-3 Robustness (3 seeds)")
plt.xticks(rotation=20, ha='right')
plt.tight_layout()
plt.savefig("top3_robustness_mean_std.png", dpi=150, bbox_inches="tight")
plt.show()
print("✓ Saved robustness plot -> top3_robustness_mean_std.png")
```

```
================================================================================
EXPERIMENT top3_Drop0.5_2L_64U_seed1: Drop0.5_2L_64U
================================================================================
Parameters: 644,289
✓ Test Acc: 0.8832 | AUC: 0.9507 | Best Val Acc: 0.8896 | Time: 22.4s
Confusion Matrix:
 [[11072  1428]
 [ 1491 11009]]
              precision    recall  f1-score   support

         0.0      0.881     0.886     0.884     12500
         1.0      0.885     0.881     0.883     12500

    accuracy                          0.883     25000
   macro avg      0.883     0.883     0.883     25000
weighted avg      0.883     0.883     0.883     25000


================================================================================
EXPERIMENT top3_Drop0.5_2L_64U_seed2: Drop0.5_2L_64U
================================================================================
Parameters: 644,289
✓ Test Acc: 0.8839 | AUC: 0.9507 | Best Val Acc: 0.8879 | Time: 19.1s
Confusion Matrix:
 [[10929  1571]
 [ 1332 11168]]
              precision    recall  f1-score   support

         0.0      0.891     0.874     0.883     12500
         1.0      0.877     0.893     0.885     12500

    accuracy                          0.884     25000
   macro avg      0.884     0.884     0.884     25000
weighted avg      0.884     0.884     0.884     25000


================================================================================
EXPERIMENT top3_Drop0.5_2L_64U_seed3: Drop0.5_2L_64U
================================================================================
Parameters: 644,289
✓ Test Acc: 0.8841 | AUC: 0.9507 | Best Val Acc: 0.8882 | Time: 21.9s
Confusion Matrix:
 [[10947  1553]
 [ 1344 11156]]
              precision    recall  f1-score   support

         0.0      0.891     0.876     0.883     12500
         1.0      0.878     0.892     0.885     12500

    accuracy                          0.884     25000
   macro avg      0.884     0.884     0.884     25000
weighted avg      0.884     0.884     0.884     25000


================================================================================
EXPERIMENT top3_Drop0.3_2L_64U_seed1: Drop0.3_2L_64U
================================================================================
Parameters: 644,289
✓ Test Acc: 0.8822 | AUC: 0.9497 | Best Val Acc: 0.8898 | Time: 18.2s
Confusion Matrix:
 [[11213  1287]
 [ 1658 10842]]
              precision    recall  f1-score   support

         0.0      0.871     0.897     0.884     12500
         1.0      0.894     0.867     0.880     12500

    accuracy                          0.882     25000
   macro avg      0.883     0.882     0.882     25000
weighted avg      0.883     0.882     0.882     25000


================================================================================
EXPERIMENT top3_Drop0.3_2L_64U_seed2: Drop0.3_2L_64U
================================================================================
Parameters: 644,289
✓ Test Acc: 0.8835 | AUC: 0.9503 | Best Val Acc: 0.8882 | Time: 18.2s
Confusion Matrix:
 [[11074  1426]
 [ 1486 11014]]
              precision    recall  f1-score   support

         0.0      0.882     0.886     0.884     12500
```

```
         1.0      0.885     0.881     0.883     12500

    accuracy                          0.884     25000
   macro avg      0.884     0.884     0.884     25000
weighted avg      0.884     0.884     0.884     25000


================================================================================
EXPERIMENT top3_Drop0.3_2L_64U_seed3: Drop0.3_2L_64U
================================================================================
Parameters: 644,289
✓ Test Acc: 0.8810 | AUC: 0.9490 | Best Val Acc: 0.8881 | Time: 26.8s
Confusion Matrix:
 [[11057  1443]
 [ 1531 10969]]
              precision    recall  f1-score   support

         0.0      0.878     0.885     0.881     12500
         1.0      0.884     0.878     0.881     12500

    accuracy                          0.881     25000
   macro avg      0.881     0.881     0.881     25000
weighted avg      0.881     0.881     0.881     25000


================================================================================
EXPERIMENT top3_2L_64U_ReLU_seed1: 2L_64U_ReLU
================================================================================
Parameters: 644,289
✓ Test Acc: 0.8857 | AUC: 0.9505 | Best Val Acc: 0.8888 | Time: 17.5s
Confusion Matrix:
 [[10984  1516]
 [ 1341 11159]]
              precision    recall  f1-score   support

         0.0      0.891     0.879     0.885     12500
         1.0      0.880     0.893     0.887     12500

    accuracy                          0.886     25000
   macro avg      0.886     0.886     0.886     25000
weighted avg      0.886     0.886     0.886     25000


================================================================================
EXPERIMENT top3_2L_64U_ReLU_seed2: 2L_64U_ReLU
================================================================================
Parameters: 644,289
✓ Test Acc: 0.8811 | AUC: 0.9474 | Best Val Acc: 0.8872 | Time: 23.6s
Confusion Matrix:
 [[11137  1363]
 [ 1609 10891]]
              precision    recall  f1-score   support

         0.0      0.874     0.891     0.882     12500
         1.0      0.889     0.871     0.880     12500

    accuracy                          0.881     25000
   macro avg      0.881     0.881     0.881     25000
weighted avg      0.881     0.881     0.881     25000
```

Next steps: Generate code with top3_stats New interactive sheet

```
================================================================================
EXPERIMENT top3_2L_64U_ReLU_seed3: 2L_64U_ReLU
================================================================================
Parameters: 644,289
✓ Test Acc: 0.8804 | AUC: 0.9494 | Best Val Acc: 0.8901 | Time: 19.6s
Confusion Matrix:
 [[10759  1741]
 [ 1249 11251]]
              precision    recall  f1-score   support

         0.0      0.896     0.861     0.878     12500
         1.0      0.866     0.900     0.883     12500

    accuracy                          0.880     25000
   macro avg      0.881     0.880     0.880     25000
weighted avg      0.881     0.880     0.880     25000
```

	name	mean_test_acc	std_test_acc	mean_auc	std_auc	mean_train_time_s
0	Drop0.5_2L_64U	0.883747	0.000371	0.950708	0.000024	29.578135
1	Drop0.3_2L_64U	0.882253	0.001013	0.949666	0.000549	30.134721
2	2L_64U_ReLU	0.882413	0.002357	0.949102	0.001299	28.896224

```
✓ Saved robustness table -> top3_robustness_stats.csv
```

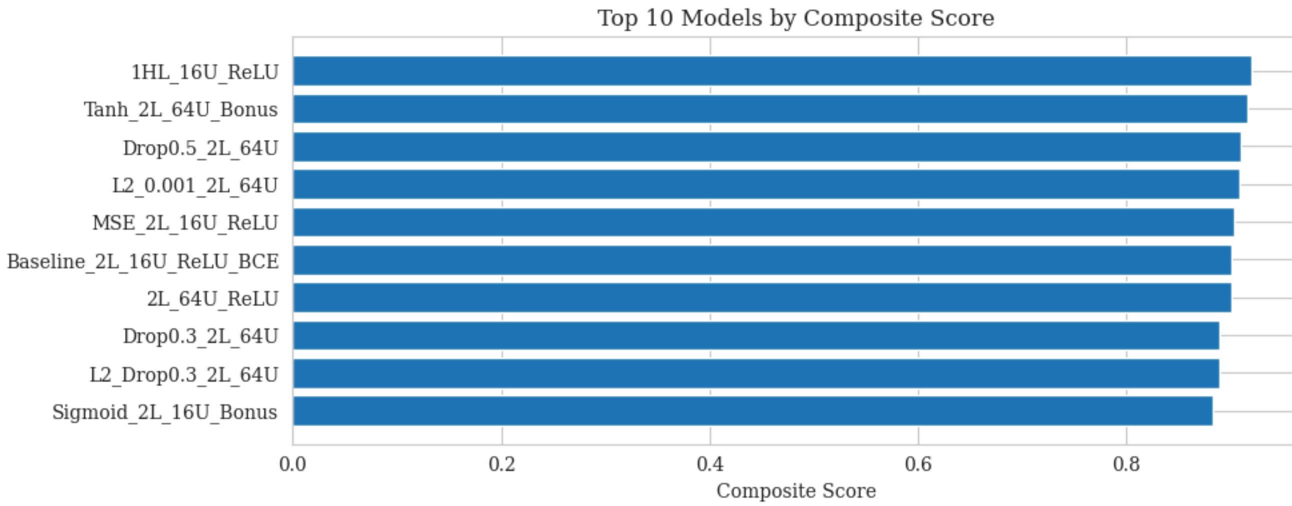
Compare all models, recommend best, and visualize

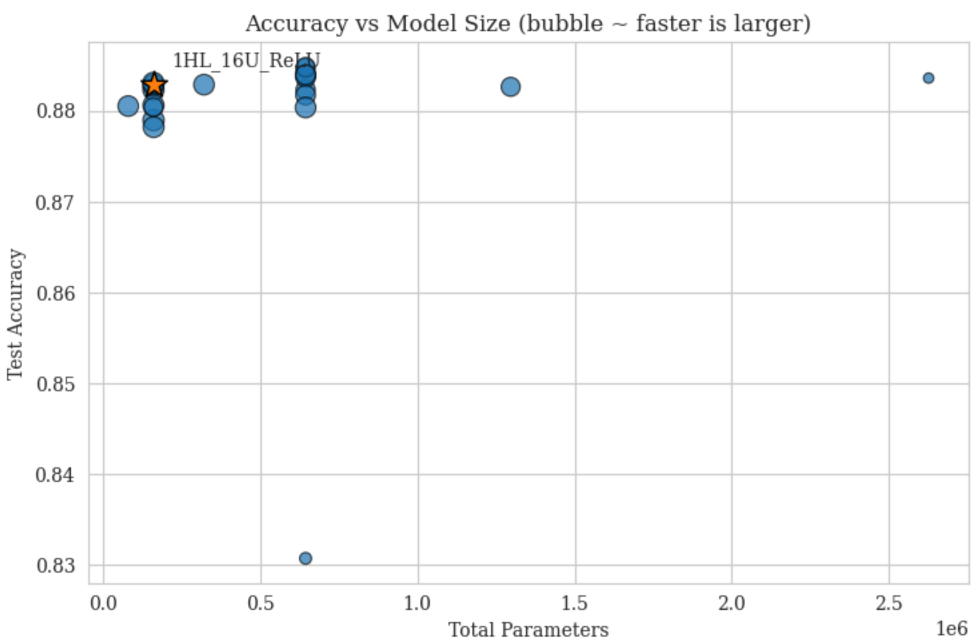

```
# ====  Compare all models, recommend best, and visualize ====

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from pathlib import Path

# 1) Load results if needed
if 'df' not in globals():
    if Path("full_results_final.csv").exists():
        df = pd.read_csv("full_results_final.csv")
    else:
        raise RuntimeError("No df in memory and full_results_final.csv not found.")

# 2) Ensure needed columns exist; fill safe defaults if missing
need_cols = {
    "model_name": "model_name",
    "test_accuracy": "test_accuracy",
    "auc": "auc",
    "overfitting_gap": "overfitting_gap",
    "total_params": "total_params",
    "training_time": "training_time",
}
for k, v in need_cols.items():
    if v not in df.columns:
        # fill neutral defaults if absent
        if v in ("overfitting_gap", "training_time", "total_params"):
            df[v] = df.get(v, pd.Series([np.nan]*len(df))).fillna(df[v].median() if df[v].notna().any() else 0.0)
        elif v in ("test_accuracy", "auc"):
            df[v] = df.get(v, pd.Series([np.nan]*len(df))).fillna(0.0)
        else:
            raise RuntimeError(f"Required column '{v}' missing and cannot be defaulted.")

work = df.copy()

# 3) Build normalized metrics (0..1) with correct direction
def minmax(col, higher_is_better=True):
    x = work[col].astype(float).values
    xmin, xmax = np.nanmin(x), np.nanmax(x)
    if np.isclose(xmin, xmax):
        # constant column -> neutral 0.5
        return np.full_like(x, 0.5, dtype=float)
    z = (x - xmin) / (xmax - xmin)
    return z if higher_is_better else (1.0 - z)

work["_nz_acc"]   = minmax("test_accuracy", True)
work["_nz_auc"]   = minmax("auc", True)
work["_nz_gap"]   = minmax("overfitting_gap", False)
work["_nz_param"] = minmax("total_params", False)
work["_nz_time"]  = minmax("training_time", False)

# 4) Composite score (tweak weights if you like)
w = {
    "acc": 0.45,
    "auc": 0.25,
    "gap": 0.15,
    "param": 0.10,
    "time": 0.05,
}
work["composite_score"] = (
    w["acc"]  * work["_nz_acc"] +
    w["auc"]  * work["_nz_auc"] +
    w["gap"]  * work["_nz_gap"] +
    w["param"]* work["_nz_param"] +
    w["time"] * work["_nz_time"]
)

# 5) Rank and recommend
ranked = work.sort_values(["composite_score", "test_accuracy", "auc"], ascending=[False, False, False]).reset_index(drop=True)
best = ranked.iloc[0]

print("=== Recommendation ===")
```

```
print(f"Model: {best['model_name']}")
print(f"Composite Score: {best['composite_score']:.4f}")
print(f"Test Accuracy: {best['test_accuracy']:.4f} | AUC: {best['auc']:.4f}")
print(f"Overfitting Gap: {best['overfitting_gap']:.4f} | Params: {int(best['total_params']):,} | Train Time: {best['training_time'

# 6) Display comparison table (top 10)
cols_show = [
    "model_name", "test_accuracy", "auc", "overfitting_gap",
    "total_params", "training_time", "composite_score"
]
display(ranked[cols_show].head(10))

# 7) Visuals
topk = ranked.head(10)

# (a) Top-10 composite bar chart
plt.figure(figsize=(10, 4))
plt.barh(topk["model_name"], topk["composite_score"])
plt.gca().invert_yaxis()
plt.xlabel("Composite Score")
plt.title("Top 10 Models by Composite Score")
plt.tight_layout()
plt.savefig("compare_top10_composite.png", dpi=150, bbox_inches="tight")
plt.show()

# (b) Trade-off scatter: Params vs Test Accuracy (bubble size=time, edge shows gap)
plt.figure(figsize=(7.5, 5))
sizes = 100 * (minmax("training_time", False)) + 30  # ensure minimum size
scatter = plt.scatter(
    work["total_params"], work["test_accuracy"],
    s=sizes, alpha=0.7, linewidth=0.8, edgecolors="k"
)
plt.xlabel("Total Parameters")
plt.ylabel("Test Accuracy")
plt.title("Accuracy vs Model Size (bubble ~ faster is larger)")
# Annotate best model
plt.scatter([best["total_params"]], [best["test_accuracy"]], s=220, marker="*", edgecolors="k")
plt.annotate(best["model_name"], (best["total_params"], best["test_accuracy"]),
             xytext=(10, 10), textcoords="offset points")
plt.tight_layout()
plt.savefig("compare_acc_vs_params.png", dpi=150, bbox_inches="tight")
plt.show()

print("✓ Saved: compare_top10_composite.png, compare_acc_vs_params.png")
```

```
=== Recommendation ===
Model: 1HL_16U_ReLU
Composite Score: 0.9197
Test Accuracy: 0.8829 | AUC: 0.9495
Overfitting Gap: 0.0623 | Params: 160,033 | Train Time: 10.6s
```

	model_name	test_accuracy	auc	overfitting_gap	total_params	training_time	composite_score
0	1HL_16U_ReLU	0.88288	0.949523	0.062333	160033	10.627203	0.919712
1	Tanh_2L_64U_Bonus	0.88376	0.951218	0.058700	644289	16.273567	0.915241
2	Drop0.5_2L_64U	0.88488	0.951522	0.068167	644289	22.334092	0.909480
3	L2_0.001_2L_64U	0.88400	0.949745	0.058400	644289	17.193099	0.907889
4	MSE_2L_16U_ReLU	0.88312	0.949312	0.078300	160305	11.010390	0.903331
5	Baseline_2L_16U_ReLU_BCE	0.88236	0.949166	0.074067	160305	10.582889	0.901079
6	2L_64U_ReLU	0.88408	0.949129	0.064867	644289	15.693160	0.899911
7	Drop0.3_2L_64U	0.88476	0.950354	0.080667	644289	21.483936	0.889561
8	L2_Drop0.3_2L_64U	0.88176	0.950309	0.061467	644289	17.935163	0.888515
9	Sigmoid_2L_16U_Bonus	0.88044	0.947855	0.055800	160305	22.520850	0.882894

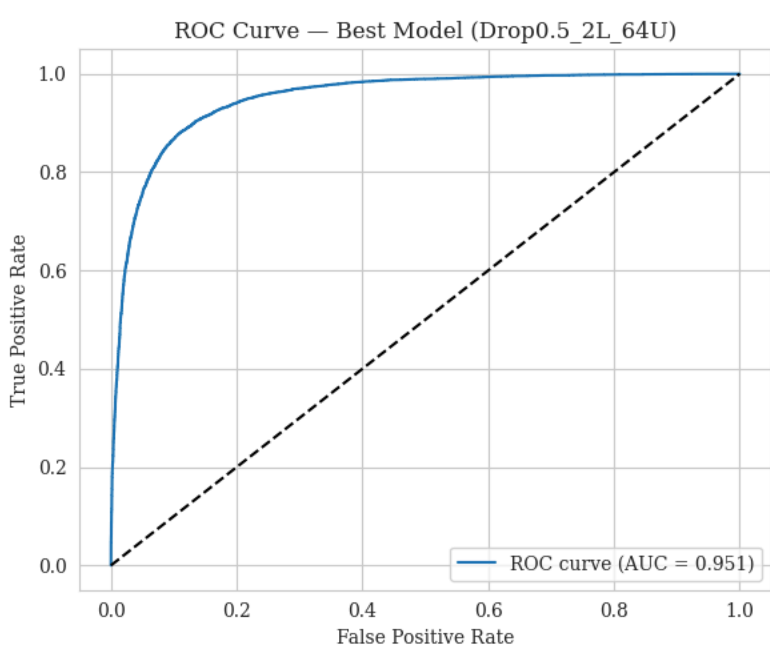





```
✓ Saved: compare_top10_composite.png, compare_acc_vs_params.png
```

ROC curve for the best model

```
# ==== ROC curve for the best model ====

from sklearn.metrics import roc_curve, auc

# Rebuild & reload the best model if needed
best_cfg = {
    "name": "Drop0.5_2L_64U",
    "num_layers": 2,
    "units": 64,
    "activation": "relu",
    "loss": "binary_crossentropy",
    "optimizer": "adam",
    "use_dropout": True,
    "dropout_rate": 0.5,
    "use_l2": True,
    "l2_strength": 0.001
}

best_model = build_model_safe(best_cfg)
best_model.load_weights("chk_Drop0.5_2L_64U.keras")

# Predict probabilities
y_prob = best_model.predict(x_test, verbose=0).ravel()

# Compute ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_test, y_prob)
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(6, 5))
plt.plot(fpr, tpr, label=f"ROC curve (AUC = {roc_auc:.3f})")
plt.plot([0, 1], [0, 1], "k--")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve — Best Model (Drop0.5_2L_64U)")
plt.legend(loc="lower right")
plt.tight_layout()
plt.savefig("roc_curve_best_model.png", dpi=150, bbox_inches="tight")
plt.show()

print(f"✓ ROC curve plotted and saved (AUC = {roc_auc:.3f})")
```



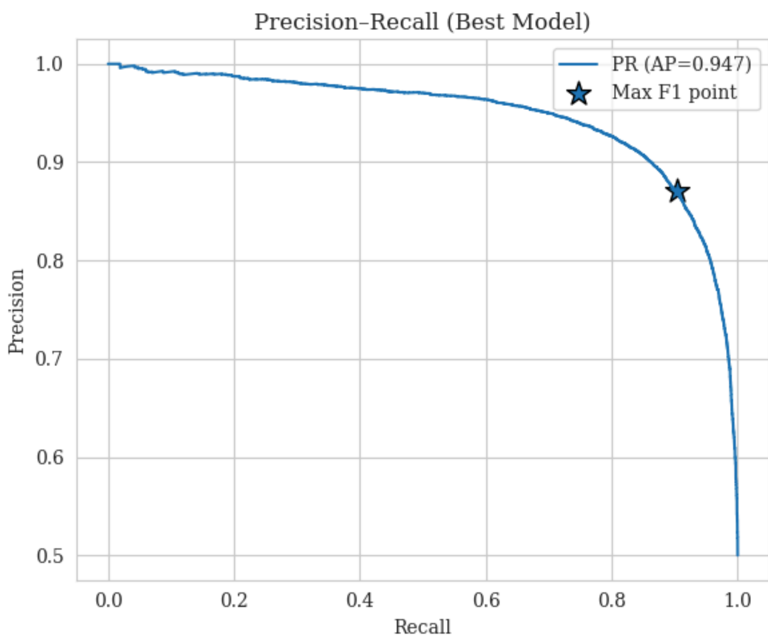

```
✓ ROC curve plotted and saved (AUC = 0.951)
```

Precision–Recall (Best Model)

```
# Optimize decision threshold for best model; add PR curve
from sklearn.metrics import f1_score, precision_recall_curve, average_precision_score
```

```
# rebuild & load best model weights if needed
best_cfg = {
    "name": "Drop0.5_2L_64U", "num_layers": 2, "units": 64, "activation": "relu",
    "loss": "binary_crossentropy", "optimizer": "adam",
    "use_dropout": True, "dropout_rate": 0.5, "use_l2": True, "l2_strength": 0.001
}
best_model = build_model_safe(best_cfg)
best_model.load_weights("chk_Drop0.5_2L_64U.keras")

y_prob = best_model.predict(x_test, verbose=0).ravel()
ts = np.linspace(0.3, 0.7, 41)
f1s = [f1_score(y_test, (y_prob >= t).astype(int)) for t in ts]
t_best = float(ts[int(np.argmax(f1s))])
f1_best = float(np.max(f1s))

prec, rec, thr = precision_recall_curve(y_test, y_prob)
ap = average_precision_score(y_test, y_prob)

print(f"✓ Best threshold by F1: {t_best:.2f} (F1={f1_best:.3f}, AP={ap:.3f})")

plt.figure(figsize=(6,5))
plt.plot(rec, prec, label=f"PR (AP={ap:.3f})")
plt.scatter([rec[np.argmax((2*prec*rec)/(prec+rec+1e-9))]],
            [prec[np.argmax((2*prec*rec)/(prec+rec+1e-9))]],
            marker="*", s=180, edgecolors="k", label="Max F1 point")
plt.xlabel("Recall"); plt.ylabel("Precision"); plt.title("Precision-Recall (Best Model)")
plt.legend(); plt.tight_layout()
plt.savefig("pr_curve_best_model.png", dpi=150, bbox_inches="tight")
plt.show()
```

```
✓ Best threshold by F1: 0.46 (F1=0.886, AP=0.947)
```

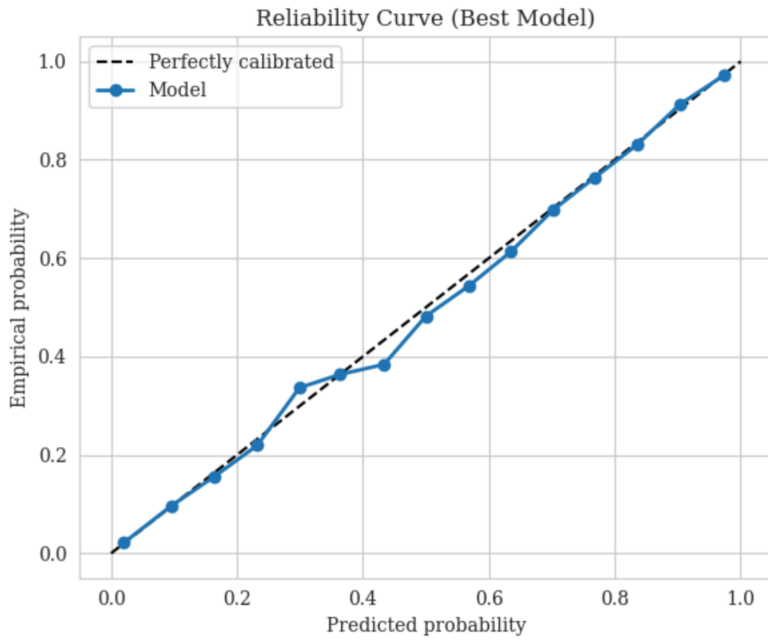


Reliability Curve (Best Model)

```
from sklearn.metrics import brier_score_loss
from sklearn.calibration import calibration_curve

brier = brier_score_loss(y_test, y_prob)
print(f"✓ Brier score (lower is better): {brier:.4f}")

prob_true, prob_pred = calibration_curve(y_test, y_prob, n_bins=15, strategy="uniform")

plt.figure(figsize=(6,5))
plt.plot([0,1],[0,1],"k--", label="Perfectly calibrated")
plt.plot(prob_pred, prob_true, marker="o", linewidth=2, label="Model")
plt.xlabel("Predicted probability"); plt.ylabel("Empirical probability")
```

```
plt.title("Reliability Curve (Best Model)")
plt.legend(); plt.tight_layout()
plt.savefig("calibration_curve_best_model.png", dpi=150, bbox_inches="tight")
plt.show()
```

```
✓ Brier score (lower is better): 0.0849
```



best vs runner up

```
from sklearn.model_selection import StratifiedKFold
from scipy.stats import wilcoxon

# choose two finalists to compare (best vs runner-up from your df)
top2 = df.sort_values("test_accuracy", ascending=False).head(2)["model_name"].tolist()
cfg_map = {r["model_name"]: r for r in results}  # results list from Cell 6

def cfg_from_row(row):
    return {
        "name": row["model_name"], "num_layers": int(row["num_layers"]),
        "units": int(row["units"]), "activation": str(row["activation"]),
        "loss": str(row["loss_function"]), "optimizer": str(row["optimizer"]),
        "use_dropout": bool(row["use_dropout"]), "dropout_rate": float(row["dropout_rate"]),
        "use_l2": bool(row["use_l2"]), "l2_strength": float(row["l2_strength"]),
    }

finalists = [cfg_from_row(df[df["model_name"]==m].iloc[0]) for m in top2]

# create a single train+val pool to do 5-fold
X = np.vstack([x_train, x_val]); y = np.hstack([y_train, y_val])
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=RANDOM_SEED)

fold_acc = {top2[0]: [], top2[1]: []}
for fold, (tr, va) in enumerate(skf.split(X, y), start=1):
    Xtr, Ytr, Xva, Yva = X[tr], y[tr], X[va], y[va]
    for mname, cfg in zip(top2, finalists):
        keras.utils.set_random_seed(RANDOM_SEED + fold)
        model = build_model_safe(cfg)
        model.fit(Xtr, Ytr, validation_data=(Xva, Yva),
                  epochs=EPOCHS, batch_size=BATCH_SIZE, verbose=0,
                  callbacks=[EarlyStopping(monitor="val_accuracy", mode="max", patience=3, restore_best_weights=True)])
        _, acc = model.evaluate(x_test, y_test, verbose=0)
        fold_acc[mname].append(acc)
        print(f"Fold {fold} | {mname}: test_acc={acc:.4f}")

acc_a = np.array(fold_acc[top2[0]]); acc_b = np.array(fold_acc[top2[1]])
stat_p = wilcoxon(acc_a, acc_b, alternative="greater")  # is top2[0] > top2[1]?
print(f"\n✓ 5-fold test accuracies:\n{top2[0]}: {acc_a.mean():.4f} ± {acc_a.std():.4f}\n{top2[1]}: {acc_b.mean():.4f} ± {acc_b.st
```