Antonio Augusto Pilan dos Santos
William de Lima Anselmo

Introduction to machine learning project

An introduction to deep learning with binary classification of soybean proximal images

Ribeirão Preto, 2025
Professor José Augusto Baranauskas

Abstract

This work addresses the use of deep learning techniques to automate the classification process of healthy and unhealthy soybean grains, a relevant topic for Brazilian agribusiness, as the country is one of the world's largest soybean producers. Despite advances in genetic improvement, soybean cultivation still faces significant losses caused by diseases, pests, and climate change. As an alternative to manual monitoring, which is unfeasible on a large scale, the study proposes the use of computer vision applied to grain image analysis, using a public dataset produced and published as an article. The images underwent segmentation processes and were labeled as healthy or unhealthy. However, the dataset presents an imbalance, with approximately 78% of the samples being unhealthy grains, which requires the use of metrics such as recall and precision to evaluate the model's performance. The project uses TensorFlow in Python and is aimed at beginner students in graduate and undergraduate programs. The chosen method was Deep Learning, due to its ability to handle complex and non-linear data without relying heavily on manual feature engineering. This work's purpose is to build a step-by-step Machine Learning process, considering potential issues and building a model that is capable of generalizing in an imbalanced dataset. With 77.39% recall and 83.71% accuracy in a dataset that has less than 30% of the samples being the one we want to identify, is it enough? *how can it get better?*

Keywords: Machine Learning, Data Science, Classification Models, Deep Learning.

1 Introduction

Agribusiness is one of the main economic and technological drivers of Brazil and, especially relevant for our work, the country is responsible for 40% of the soybean produced worldwide [6], being one of the most important agricultural commodities in the world, widely used for both animal feed production and human consumption, as well as industrial applications. However, despite advances in genetic improvement for developing more resistant varieties, soybean production still faces significant challenges, mainly in field management.

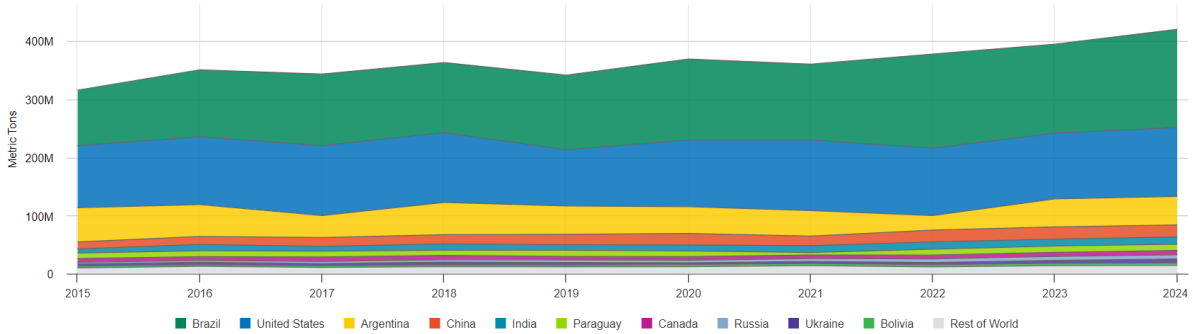


Figure 1: Soybean production in different countries in tons produced annually, USDA chart [6]

Large-scale factors such as diseases, pests, and climate change can negatively impact yields, resulting in losses exceeding 20% of global production [1]. Given this complicated scenario for soybean productivity, efficient crop monitoring becomes essential to minimize negative impacts and optimize production. However, conducting this monitoring manually requires a large workforce, which may be unfeasible on a large scale.

In this context, the use and application of computer vision techniques emerge as a promising alternative that allows the automation of analyzing critical variables for soybean cultivation.

With advances in artificial intelligence, especially in deep learning, we can use images directly from the field for various applications that have been explored in agriculture, ranging from microscales such as disease recognition to estimating crop yield. Although some of these approaches already show promising results, there are many technical challenges such as data variability and representativeness, the need for large volumes of information for training models, and the difficulty of generalizing solutions, since real conditions found in the field can vary in unpredictable ways.

Therefore, it is possible to say that these technologies, despite their growth, are little used in the field in practice since implementation requires scientific improvements. Given this scenario, this is an introductory work aimed at students recently admitted to graduate programs and more advanced undergraduate students that aims to understand the practical application of machine learning techniques in soybean management from the analysis of proximal images of soybean grain, detecting whether that grain is a healthy or unhealthy grain, seeking, mainly, to understand and visualize the generalization of a model and the machine learning process to build it.

1.1 Soybean grain quality

1.1.1 The dataset

The dataset used was published in 2023 by several authors (Lin et al.) and its methodology, as well as the authors, is documented in the article *Soybean image dataset for classification* [5]. These are proximal images of soybean grains that underwent treatment to isolate the soybean grain, as shown in image 2

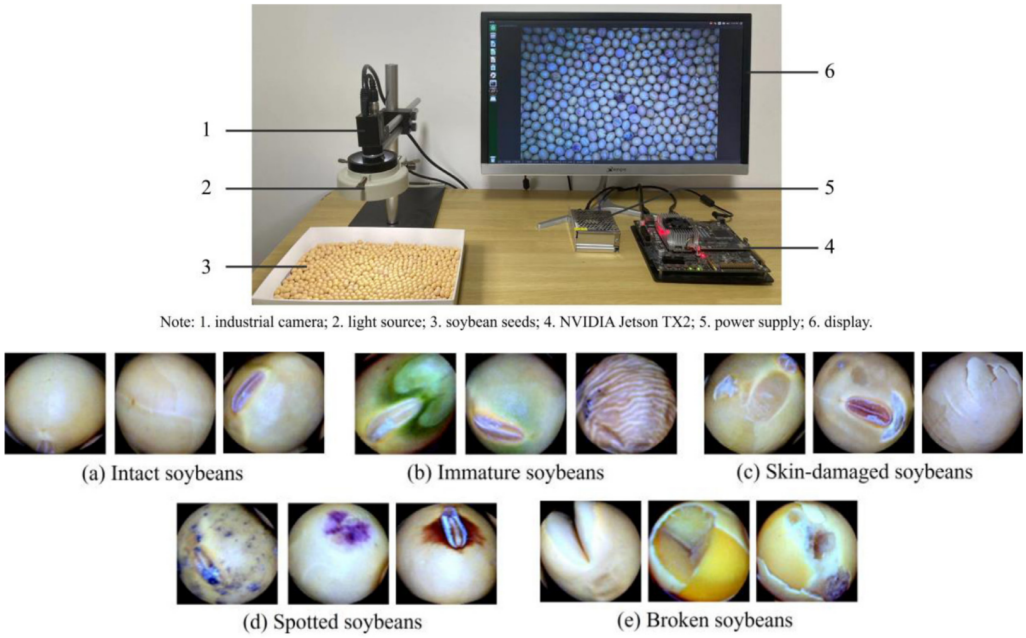


Figure 2: Dataset generation scheme [5]

A photo is taken of a large set of seeds and, using a high-pass filter and image segmentation algorithms, the grains are automatically selected by the dataset authors, and finally, the images that we will use in building our model are generated. Regarding the generation of images, labeling, and all the details of the dataset, I recommend reading the article that describes the complete process of consolidating the database [5].

As a result, we have several soybean grain images that were labeled with their *targets* for model creation. Specifically for this work, we will perform a binary classification between healthy and unhealthy grains, simplifying the database to only 2 *targets*.

Every choice is a renunciation. By choosing to simplify the classification process to just two classes, we introduce a high imbalance in the data. The dataset contains approximately 78% of the data categorized as *unhealthy* as can be seen in table 1. Therefore, models tend to choose this label in prediction, since the probability of being correct is higher. Therefore, we need to verify the model's performance using normalized metrics such as *recall*, precision and f1-score.

Table 1: Target schema

Class	No. of Instances
Healthy	1201
Unhealthy	4312

2 Methodology

Complex problems require robust solutions. One of the biggest problems presented in Jayme Garcia's article [1] is the generalization of training data. Convolutional Neural Networks (CNN) tend to be an appropriate choice to deal with non-linearity in high dimensional spaces such as raw image data. Therefore, we will resort to simple Deep Learning CNN structure model at an introductory level, given the audience to which this article is directed.

The project will treat the data as tensors using *TensorFlow* in the *Python* library, using as reference the book *Deep Learning With Python* by François Chollet [2].

2.1 Deep Learning

We will use Deep Learning as our engine to identify healthy soybean grains. The technique stands out, mainly, for being able to automate the process of *feature engineering* and work with complex data distributions that classical models would struggle classifying.

We will work with **tensors** in the TensorFlow library and the weights and *biases* will be optimized by minimizing a cost function that evaluates binary classifications.

3 Theory and coding

3.1 Theory

3.1.1 The layered structure of a Deep Learning Model

When we think about Deep Learning models, the first thing that should come to our minds is its layered nature. It is the core of Deep Learning and for the frameworks that are built for developing AI models.

We see the layers as mathematical operations. It takes an input, makes its thing and then outputs "something else". In general words, it executes a space transformation and the output is the mapped equivalent of the input in the new space.

Overall, this *mapping* is just like fitting a curve. Natural data lies in a high dimensional structure that can be mapped. This is called the Manifold Hypostesis, in which high dimensional

data lies in a lower dimension manifold that can be fitted. So it can be learned by adjusting the fitting parameters.

It happens similarly as fitting a linear function $y = f(x)$. If our model need to predict points on the line, we just need to understand what parameters β and α are.

$$y = \alpha x + \beta$$

In this example, our data lies in a high dimensional structure that can be mapped through $y = f(x)$. In that way, we're allowed to estimate future values that have never been seen by our model as exemplified on figure 3.

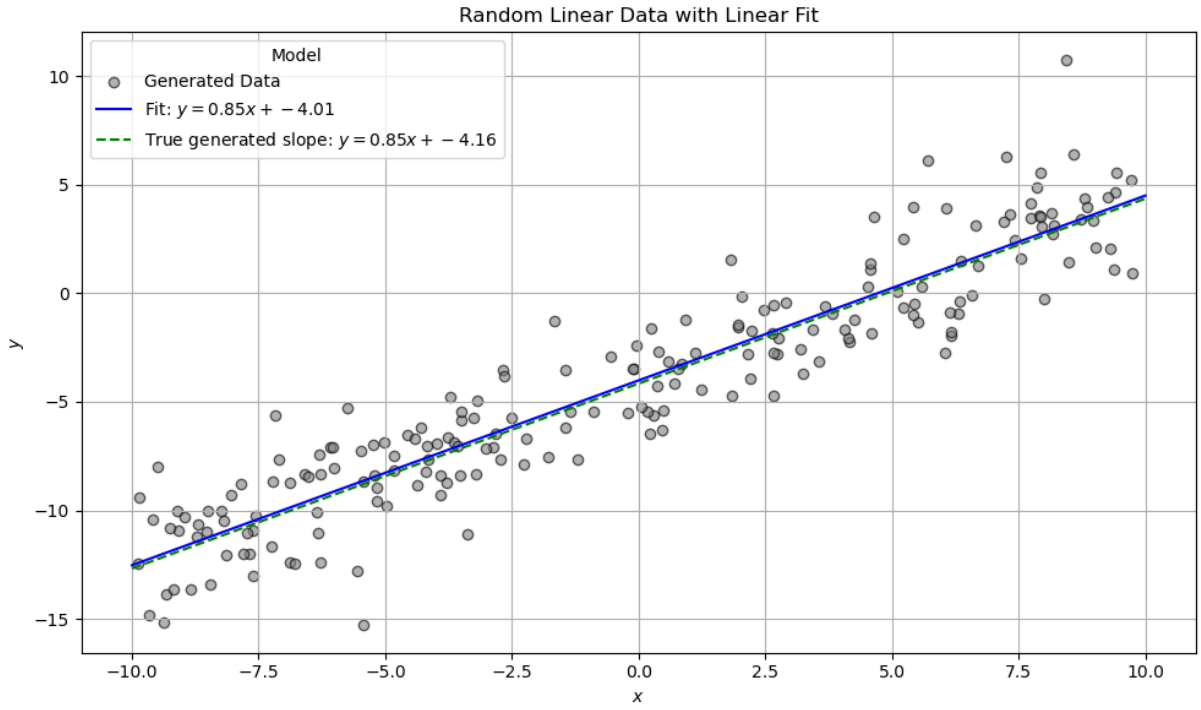


Figure 3: Example of linear fitting with random generated data. All data samples were generated with a parameterized line adding noise for inducing variance. A linear fitting of this example of "natural data" can illustrate the simplest imaginable machine learning problem

Now, for more complex and higher-dimensional problems, think of our model as repeated over and over for many layers. Each layer is responsible for contributing to a more complex mapping into a complex output, until we get to the point where soybean grain images can be classified into healthy or unhealthy. A very complex manifold with more dimensions that we can't even picture in our minds can be *"fitted"* to its parametric form. And that's it, *"deep learning is curve fitting, not magic"* - François Chollet[2].

3.1.2 Loss function

But how to tune those parameters in a very complex model? Well, we first need to evaluate how much of our model's output is correct. We do it by using a **loss function**. It is basically a method of calculating the *"distance"* between model's output and the validation set, a Δy that measures how much $y_{predicted}$ differs from y_{true} . In many cases, literal distance metrics can be used such as Euclidean distance, but in our binary classification problem a more categorical metric is needed, and that is *binary cross-entropy*.

This method basically works the difference of *Shannon's Entropy* and its complementary balanced out from the true values of y as follows [4]:

$$H_{BCE} = -\frac{1}{N}\sum y_i \log p_i + (1 - y_i) \log 1 - p_i$$

Where y_i is the i^{eth} true value and p_i is the respective probability of the class being True.

This mathematical procedure basically balances the probability of the possible events. We can see in table 2 how it mathematically works as $y_{predicted} = y_{true}$ the logarithmic terms balance themselves out.

Table 2: Behavior of Binary Cross-Entropy (BCE) Loss Based on True and Predicted Values

y_{true}	$y_{predicted}$ = 1	$y_{predicted}$ = 0
1	$H_{BCE} \to 0$	$H_{BCE} \to \infty$
0	$H_{BCE} \to \infty$	$H_{BCE} \to 0$

So if $y_{predicted} = y_{true}$, Binary Cross-Entropy(BCE) tends to 0. In that way, this turns out to be a minimization problem, where using derivatives turns out to be pretty interesting.

3.1.3 *Make it learn*: Gradient Descent and Backpropagation

Dealing with a minimization problem gives us calculus tools. If we want to minimize our loss function that maps the input space, we can analyze the first derivative of the loss function as we change our parameters. To make it more visual, using the linear example from figure 3, if we don't know α and β parameters we can make adjustments by looking to $\frac{dH_{BCE}}{d\alpha}$.

In a high dimensional space this derivative is a gradient, we can mathematically write it as ∇H_{BCE} for our Binary Cross-Entropy problem, and moving our parameters to the negative side of our gradient recursively it may guide us to minimize our loss function. This method is called *the Gradient Descent* and each iteration it moves our loss function to a local minima as image 4 shows.

So we can "walk" through our loss function curve in a learning rate γ, that respects the gradient direction, so the next position for our loss function $H_{BCE_{i+1}}$ should be adjusted by the learning rate in the opposite direction of its gradient [3]:

$$H_{BCE_{i+1}} = H_{BCE_i} - \gamma \nabla H_{BCE_i}$$

This can basically be interpreted as an implementation of Euler's method to minimize the loss function, where the gradient is given in relation to parameters that are used for outputting $y_{predict}$, which define our loss function. So now we know which **direction** and **magnitude** we should adjust our parameters to minimize our loss function, but it tells us nothing about **how** we should do it. That's where *backpropagation* comes into stage.

For better understanding, we'll define backpropagation as a reverse engineering process: We know the final derivative that shows us the global step we need to take to minimize our loss function. So, we can decompose this derivative for each layer using the chain rule from the end to the beginning, propagating this derivative backward through our model.

Those are the fundamental steps of learning, now we'll start to understand how they'll apply in our work specifically.

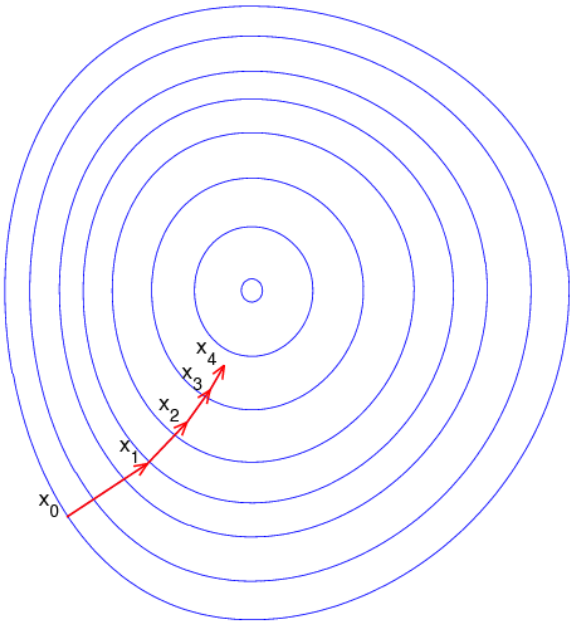


Figure 4: Gradient descent visualization[3]

3.1.4 Tensorflow, Keras and library encapsulation

Those concepts are **really** important to understand what we'll do, but the underlying math will be hidden by the main framework we'll use: Keras.

Keras is basically a Deep Learning framework that uses TensorFlow, which is a tensor operation library. All our data will be structured as n dimensional tensors and all mathematical operations are enclosed by Keras.

3.2 Methods

3.2.1 Dataset Tensor

The dataset is loaded as a tensor using the Keras interface, looping through all folders that contain the images and converting overall label in *"healthy"* or *"unhealthy"*

```
def load_and_preprocess_images(image_dir, grayscale = True):
  images = []
  labels = []


  for folder_name in os.listdir(image_dir): # acess folders on Google Drive Directory
    folder_path = os.path.join(image_dir, folder_name)
    if os.path.isdir(folder_path):

      for filename in os.listdir(folder_path): #read images inside each folder
        if filename.lower().endswith(('.png', '.jpg', '.jpeg')):
          img_path = os.path.join(folder_path, filename)
          try:
              img = Image.open(img_path)
              img = img.resize((224, 224)) # Resize to a consistent size

              if grayscale == True:
              #transform to greyscale
                  img = img.convert('L')
                  img = np.array(img)
                  img = img.reshape(224, 224, 1)
```

```
            img = tf.keras.preprocessing.image.img_to_array(img)
            img = img / 255.0 # Normalize pixel values

            if folder_name == "Intact soybeans":
              label = "healthy"
            else:
              label = "unhealthy"

            images.append(img)
            labels.append(label)
        except Exception as e:
            print(f"Error processing image {img_path}: {e}")
            continue
return tf.convert_to_tensor(images, dtype=tf.float32), tf.convert_to_tensor(labels, dtype=tf
  .string)
```

Code Snippet 1: Loading images to Jupyter's enviroment

All code and exploration is in the attached session, so from now on, only fundamental code is shown.

For sake of understanding our dataset, we can see image 5

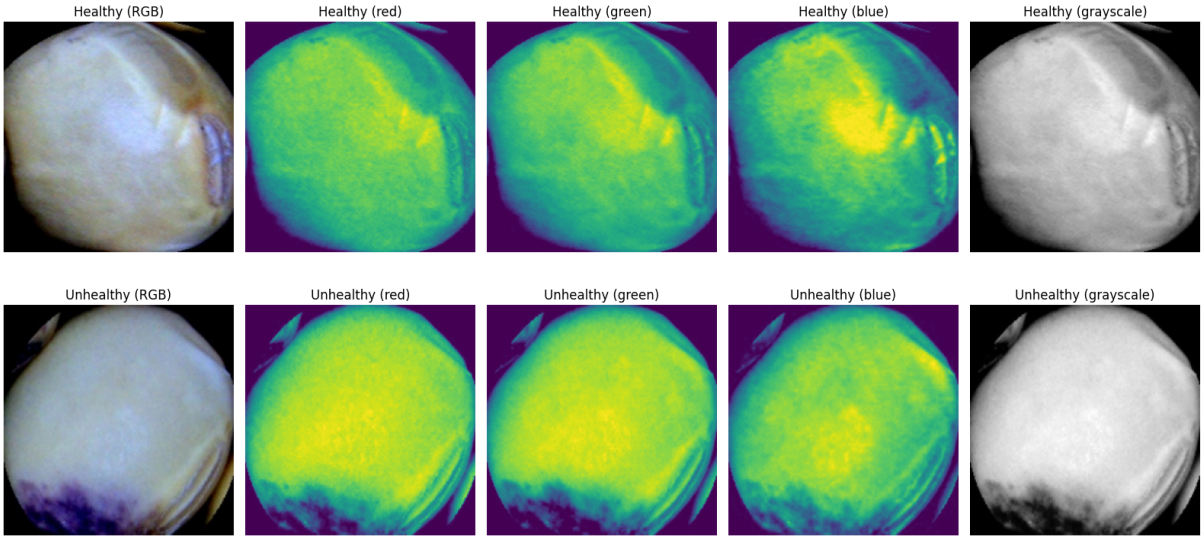


Figure 5: Seeds per channel visualization

As a introductory level, we'll feed only grayscale channel to our model and understand its learning process and some hyperparameter optimization. This choice has its caveats. Considering only the grayscale channel, we'll consider shape information, but discard color. Seeds that are not mature tends to have similar shape to healthy ones, but they have difference in color. This will not be considered by our model at first.

The end of loading data step will be splitting our data in a training/test set. Well train our model with 40% of our data for computational reasons. Every time we deal with training process, we'll split this training set in training and validation sets for, at the end of the entire exploration, we use our built model in the 60% data we allocated for testing.

3.2.2 Convolutional Neural Networks

We'll work with a basic and introductory model since our main goal is to understand its learning process and not to focus on its architecture. Code 2 shows how layers are implemented in Keras, our model will be fixed and we'll explore the training process.

In a real world scenario with more computational power, the ideal would be to optimize model's learning rate to find a optimal local minima for the loss function and optimize the batch size for a better total data representation, but due to computational limitation of this work we decided on a "good enough" batch size and a "small enough" learning rate determined by trial in the study process. We advise you, as a learner, to play with these parameters as well in your development to evaluate how this impacts the learning of the model.

It's a very common implementation of alternating Convolutional Neural Networks with 2D Pooling for overfitting control.

```
def create_model(X_train, y_train, X_test=None, y_test=None, class_weight=None, epochs = 200):
    model_convolutional = tf.keras.models.Sequential([
      Conv2D(128, (3, 3), activation='relu', input_shape=(224, 224, 1)),
      MaxPooling2D((2, 2)),
      Conv2D(64, (3, 3), activation='relu'),
      MaxPooling2D((2, 2)),
      Conv2D(32, (3, 3), activation='relu'),
      MaxPooling2D((2, 2)),
      Flatten(),
      Dense(32, activation='relu'),
      Dense(1, activation='sigmoid') # Output layer for binary classification
      ])

    metrics = [
        keras.metrics.F1Score(name="f1score"),
        keras.metrics.Precision(name="precision"),
        keras.metrics.Recall(name="recall"),
        keras.metrics.Accuracy(name="accuracy")
    ]

    model_convolutional.compile(optimizer=keras.optimizers.Adam(learning_rate=0.00001),
                                loss='binary_crossentropy', metrics=metrics)

    # Train the model
    class_weight = {0: weight_for_0, 1: weight_for_1}
    if X_test is not None and y_test is not None:
      print("Simulation model")
      history_convolutional = model_convolutional.fit(
                                                      X_train, y_train,
                                                      epochs=epochs, batch_size=128,
                                                      validation_data=(X_test, y_test),
                                                      class_weight=class_weight,
                                                      verbose = 0
                                                      )
    else:
      print("Final model")
      history_convolutional = model_convolutional.fit(
                                                      X_train, y_train,
                                                      epochs=epochs, batch_size=128,
                                                      class_weight=class_weight,
                                                      verbose = 0
                                                      )

    return model_convolutional, history_convolutional

```

Code Snippet 2: Model as a function

3.2.3 One fold simulation: understanding overfitting and its bias for grayscale model

At first, we'll understand one fold to set up our main parameters and understand our model. We'll work mostly with a fixed model with batch size and learning rate in trial and error throughout model development. Mainly, we'll implement the early-stop method and optimize a threshold decision boundary with the ROC curve and its AUC.

This first strategy is to understand the model learning parameters, as figure 6 shows: We want to understand how long we can train our model at this learning rate and this batch size to

get our model to overfit. Once we get to this point, we can optimize it to stop at optimal number of training loops.

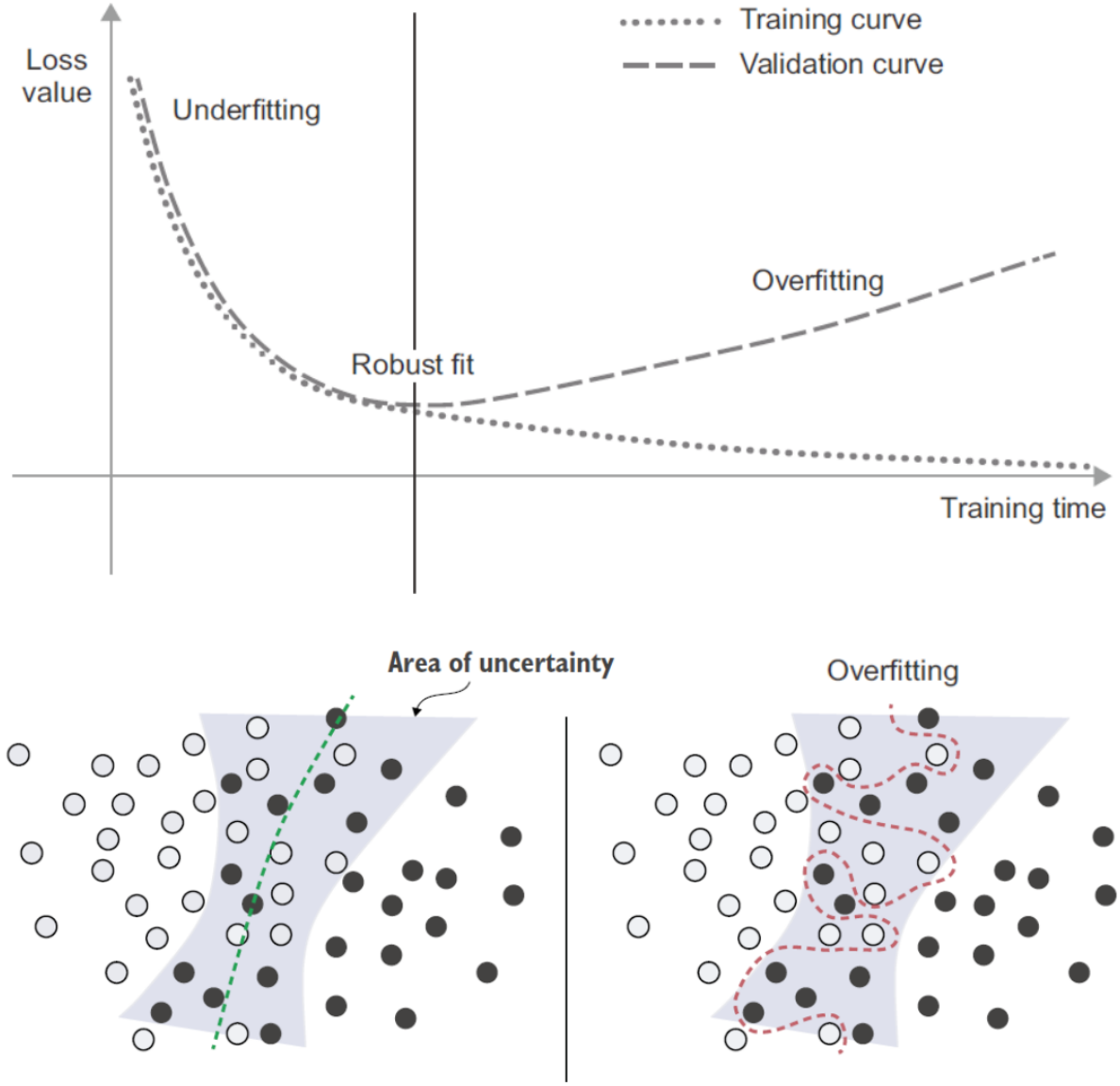


Figure 6: Overfitting example [2]. The book debates that, in manifold space, our model can *memorize* the boundary between labels in our training data, this is roughly overfitting. Generalization comes with some implied error as boundary gets more complex and undefined

3.2.4 K-fold validation

Once visualized our global scenario, it's time to do a K-fold validation and store our epochs and thresholds for us to get the best f1-score possible. We programmed a 5-fold simulation and stored all data for posterior evaluation; in our code, we used Claude AI to enhance the evaluation and visualizations on these loops.

Our best epoch will be the one with lower loss in the validation set. We'll consider it as the best possible generalization of that training/test split. So we'll finally predict a validation set and analyze AUC for different decision thresholds and set the optimal one.

4 Results and discussions

Once our model is built, we'll take a deep dive in the 1-fold simulation before going to the k-fold one.

4.1 Model Validation

4.1.1 First, we overfit!

Finding the optimal number of loops for our learning process is a process where we first need to overfit our model. On the first one-fold training, we got a good training curve and got our model to overfit as in figure 7. We got a very long training time for this specific one-fold example for us to test the limits of our model and, once it's all trained, we can evaluate different decision thresholds.

We can see a lot of noise in validation loss, it's possible this could be solvable with batch optimization, on its size and its healthy representation power. As said earlier, we took our decisions to approach a certain part of the problem which fits in our computational power available.

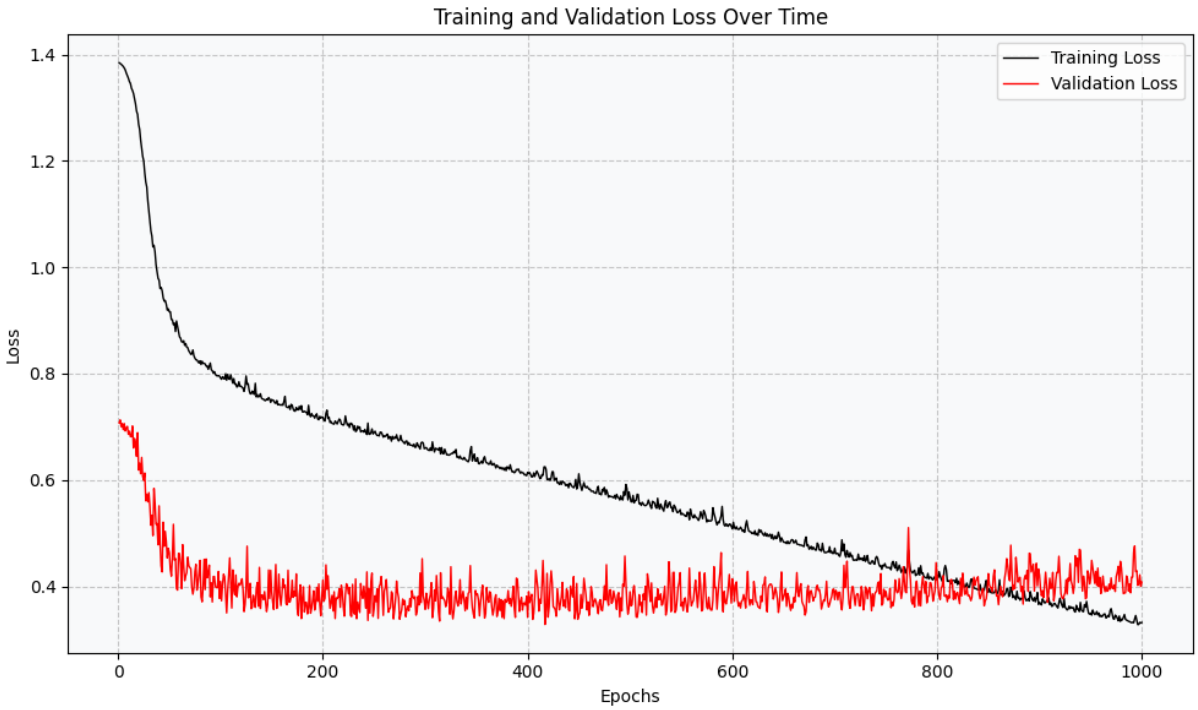


Figure 7: Training and validation loss over time results for one fold training

Now, we get a ROC curve and we decide the optimal decision threshold as stated in image 8. We see precision and recall for each decision cut, maximizing the F1-score which balances out the model's capacity to find healthy soybeans and exclude unhealthy ones.

It basically tells us that if we tell our model that it should classify a sample as **healthy** if the probability of the label is higher than the found-out threshold. In this very specific split, we got a 37.6% probability threshold that set our Recall to 87.5% and precision to 60.9%. So this one fold try should be able to identify mostly healthy seeds, but when doing it, we should insert a significant amount of unhealthy seeds on the grocery shelves.

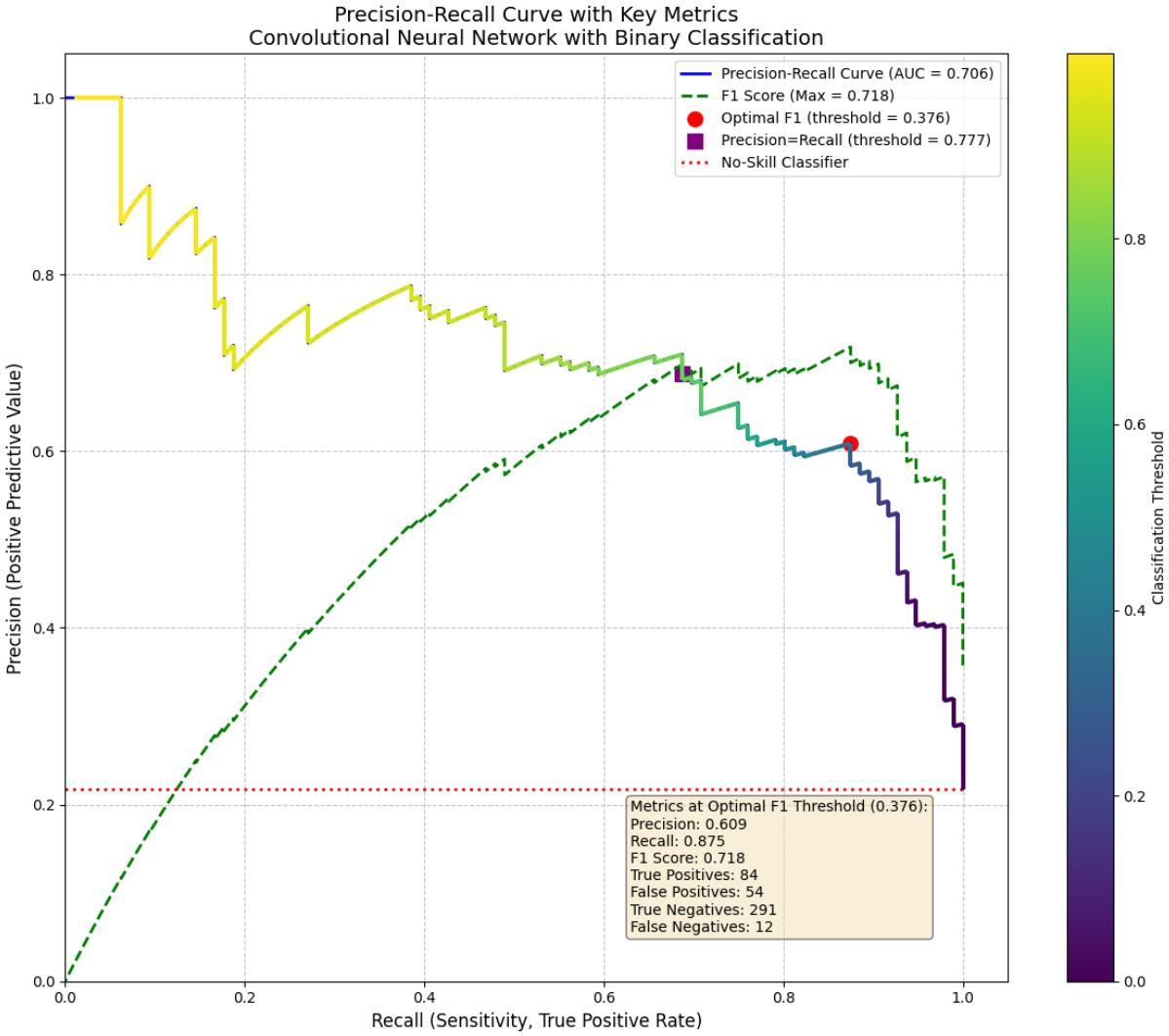


Figure 8: 1-Fold ROC curve: Our optimal threshold is the one that maximizes the f1-score

Well, this seems better than a random picking model, but not good enough for production. It should give us a hint about we can get with this structure, but lets give a k-fold approach a try to understand statistically our method.

4.1.2 Cross Validation

In a 5-fold Cross Validation, we can shuffle our train/validation split to have a more trustworthy result for our model. Table 3 shows cross-validation results. Overall, we can tell we got a very consistent Optimal Threshold and Minimal Epoch for our simulations. So we can dynamically store them in variables and run towards a final model creation using the mean value for the threshold and number of training epochs. So, finally, our model will be trained with 185 epochs and our decision of healthy seeds will happen when the probability of being healthy is bigger then $66,96\%$.

4.1.3 Final model evaluation

Finally, our final model uses just the same specifications we already shown, so this is just the icing on the cake in the entire process. We train our model with the k-fold validation optimal

Fold	F1 Score	Optimal Threshold	Minimal Epoch
1	0.688797	0.649347	174
2	0.692308	0.683708	188
3	0.686567	0.728520	184
4	0.744186	0.580238	185
5	0.682692	0.706202	197

Table 3: Cross validation folds results. See ROC curves in the attached section in k-fold validation code.

metrics and test it with test data using our optimal threshold.

Test Accuracy	83.71%
Test Precision	59.74%
Test Recall	77.39%
Test F1 Score	0.6743

Table 4: Final model testing results

The confusion matrix 9 shows us how our model performed on test data and it gets a reasonable result, but not optimal.

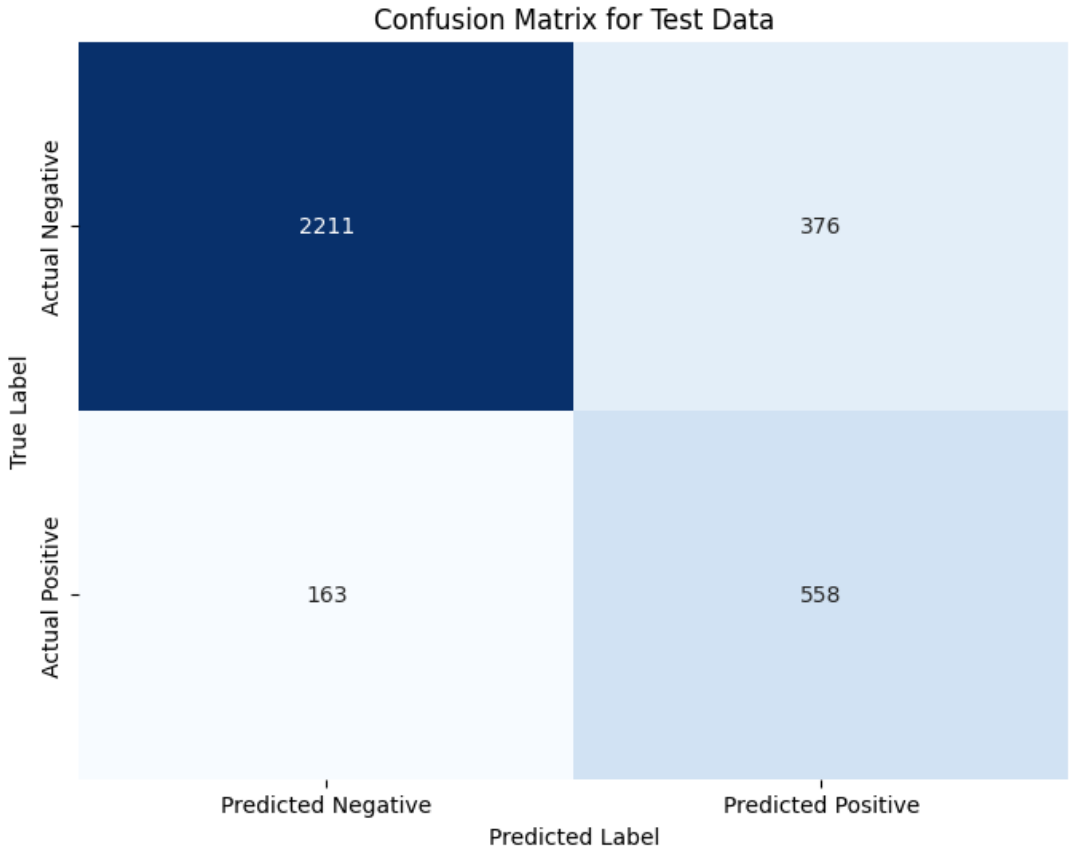


Figure 9: Final model's confusion matrix

We can see table 4 for a full report, but our model was able to have a good generalization besides its limitations. It detects good seeds and throws away most of the unhealthy ones, but, just as we expected in our initial simulations, we are still inserting a lot of unhealthy samples in

the truck for distribution. And that is what we, as scientists and engineers, need to explore: *How can it be better?*

5 Conclusion

This paper undertook an introductory exploration into the application of deep learning for the binary classification of healthy and unhealthy soybean grains, a task of considerable relevance to agricultural efficiency. The primary objective was to understand the practical implementation of a Convolutional Neural Network (CNN) on a public soybean image dataset , navigate the challenges of class imbalance, and evaluate model generalization. *We do have a structural problem*: How do we know if we got stuck in a suboptimal local minima on the hill of minimizing loss function instead of reaching a good local minima? The answer is: we don't.

While these results demonstrate a capacity to distinguish between the classes better than random chance, the precision(59.74%) indicates that a notable proportion of grains classified as healthy might be unhealthy, highlighting areas for significant improvement.

Basically, our model can be entirely underfitted and far away for an optimal minima for loss function. It may have a lot of headroom left to improve even only considering the grayscale channel. A lot of methods can further be implemented in that simple example, such as implementing RGB for color information to be considered and even more complex topics such as attention methods and feature engineering for deep learning. We can go from here to infinity as machine learning is a heavily empirical science, relying on experimentation and model optimization.

Future work should prioritize several avenues to build upon and address the identified limitations:

- **Incorporate Color Information:** Transition from grayscale to RGB color images to leverage the full spectrum of visual data.
- **Data Augmentation:** Implement data augmentation techniques (e.g., rotation, flipping, brightness adjustments) to artificially expand the diversity and effective size of the training set.
- **Advanced Imbalance Handling:** Explore more sophisticated techniques for class imbalance beyond class weighting, such as Focal Loss, or explicit oversampling of the minority class.
- **Architecture Exploration:** Experiment with varied established CNN.
- **Hyperparameter Optimization:** Conduct systematic hyperparameter tuning for learning rates, batch sizes, and optimizer parameters.
- **Refined Data Splitting:** Utilize a more conventional approach for train/test split so that the model can learn more complex patterns between samples.
- **Investigate Validation Instability:** Further analyze the causes of noisy validation loss, throughout epochs, potentially by examining the representativeness of validation folds or experimenting with different batch sizes and learning rates.
- **Feature Engineering:** The problem and dataset can be studied to a more foundational level to extract important features and give them the attention it needs.

In essence, while this project served as an introduction to the deep learning workflow for an agricultural classification task, it also highlighted the empirical and iterative nature of machine learning. The path from a foundational model to a robust, production-ready system involves continuous refinement, addressing limitations, and exploring more advanced techniques. This is what makes this science so beautiful and exciting.

References

[1] Jayme Garcia Arnal Barbedo. Deep learning for soybean monitoring and management. *Seeds*, 2023.

[2] François Chollet. *Deep Learning with Python*. Manning, second edition, 2021.

[3] Niklas Donges. What is gradient descent? `https://builtin.com/data-science/gradient-descent`. Built In. [Online; accessed 2025-05-12].

[4] GeeksforGeeks. Binary cross entropy for binary classification. `https://www.geeksforgeeks.org/binary-cross-entropy-log-loss-for-binary-classification/`. [Online; accessed 2025-05-12].

[5] Wei Lin, Youhao Fu, Peiquan Xu, Shuo Liu, Daoyi Ma, Zitian Jiang, Siyang Zang, Heyang Yao, and Qin Su. Soybean image dataset for classification. *Elsevier*, 2023.

[6] Foreign Agricultural Service. Soybeans. `https://www.fas.usda.gov/data/production/commodity/2222000`. USDA Foreign Agricultural Service [Online; accessed 2025-04-05].

Importando bibliotecas e dataset

In [1]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

In [2]:
```
import numpy as np
import pandas as pd
from scipy import interpolate
import matplotlib.pyplot as plt
from matplotlib.collections import LineCollection
from matplotlib.colors import ListedColormap, BoundaryNorm
import seaborn as sns

import os
from PIL import Image

from sklearn.model_selection import train_test_split, KFold
from sklearn.metrics import precision_recall_curve,classification_report, confus
from sklearn.metrics import average_precision_score, recall_score, f1_score, roc


import keras
import tensorflow as tf
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
```

Code Functions

In [3]:
```
def load_and_preprocess_images(image_dir, grayscale = True):
  images = []
  labels = []


  for folder_name in os.listdir(image_dir): # acess folders on Google Drive Dire
    folder_path = os.path.join(image_dir, folder_name)
    if os.path.isdir(folder_path):

      for filename in os.listdir(folder_path): #read images inside each folder
        if filename.lower().endswith(('.png', '.jpg', '.jpeg')):
          img_path = os.path.join(folder_path, filename)
          try:
              img = Image.open(img_path)
              img = img.resize((224, 224)) # Resize to a consistent size

              if grayscale == True:
              #transform to greyscale
                  img = img.convert('L')
                  img = np.array(img)
                  img = img.reshape(224, 224, 1)

              img = tf.keras.preprocessing.image.img_to_array(img)
              img = img / 255.0 # Normalize pixel values
```

```
            if folder_name == "Intact soybeans":
              label = "healthy"
            else:
              label = "unhealthy"

            images.append(img)
            labels.append(label)
        except Exception as e:
            print(f"Error processing image {img_path}: {e}")
            continue
  return tf.convert_to_tensor(images, dtype=tf.float32), tf.convert_to_tensor(la


def split_data(image_tensor, label_tensor, test_size=0.2):
    X = np.array(image_tensor)
    y = np.array(label_tensor)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_siz
    return X_train, X_test, y_train, y_test

def transform_labels(y):
    transformed_y = np.where(y == b'healthy', 1, 0)
    return transformed_y

def plot_learning_process(history):
    history_dict = history.history
    loss_values = history_dict["loss"]
    val_loss_values = history_dict["val_loss"]
    epochs = range(1, len(loss_values) + 1)
    plt.figure(figsize=(10, 6))
    plt.plot(epochs, loss_values, color='black', linewidth=1, label='Training Lo
    plt.plot(epochs, val_loss_values, color='red', linewidth=1, label='Validatio
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Training and Validation Loss Over Time')
    plt.grid(True, linestyle='--', alpha=0.7)
    plt.legend()
    plt.gca().set_facecolor('#f8f9fa')
    plt.tight_layout()
    plt.show()

def plot_precision_recall_metrics(history):
    history_dict = history.history
    train_precision = history_dict.get("precision", history_dict.get("precision_
    val_precision = history_dict.get("val_precision", history_dict.get("val_prec

    train_recall = history_dict.get("recall", history_dict.get("recall_1", []))
    val_recall = history_dict.get("val_recall", history_dict.get("val_recall_1",

    epochs = range(1, len(train_precision) + 1)

    plt.figure(figsize=(16, 6))

    plt.subplot(1, 2, 1)
    plt.plot(epochs, train_precision, color='black', linewidth=1, label='Trainin
    plt.plot(epochs, val_precision, color='red', linewidth=1, label='Validation
    plt.xlabel('Epochs')
    plt.ylabel('Precision')
    plt.title('Training and Validation Precision Over Time')
    plt.grid(True, linestyle='--', alpha=0.7)
    plt.legend()
```

```
    plt.gca().set_facecolor('#f8f9fa')

    plt.subplot(1, 2, 2)
    plt.plot(epochs, train_recall, color='black', linewidth=1, label='Training R
    plt.plot(epochs, val_recall, color='red', linewidth=1, label='Validation Rec
    plt.xlabel('Epochs')
    plt.ylabel('Recall')
    plt.title('Training and Validation Recall Over Time')
    plt.grid(True, linestyle='--', alpha=0.7)
    plt.legend()
    plt.gca().set_facecolor('#f8f9fa')

    plt.tight_layout()
    plt.show()

def plot_auc_metrics(history):
    history_dict = history.history

    train_auc = history_dict.get("AUC", history_dict.get("auc", []))
    val_auc = history_dict.get("val_AUC", history_dict.get("val_auc", []))

    epochs = range(1, len(train_auc) + 1)

    plt.figure(figsize=(10, 6))

    plt.plot(epochs, train_auc, color='black', linewidth=1, label='Training AUC'
    plt.plot(epochs, val_auc, color='red', linewidth=1, label='Validation AUC')

    plt.xlabel('Epochs')
    plt.ylabel('AUC')
    plt.title('Training and Validation AUC Over Time')
    plt.grid(True, linestyle='--', alpha=0.7)
    plt.legend()
    plt.gca().set_facecolor('#f8f9fa')

    plt.tight_layout()
    plt.show()

def plot_recall_metrics(history):
    history_dict = history.history

    train_recall = history_dict.get("recall", history_dict.get("recall", []))
    val_recall = history_dict.get("val_recall", history_dict.get("val_recall", [

    epochs = range(1, len(train_recall) + 1)

    plt.figure(figsize=(10, 6))

    plt.plot(epochs, train_recall, color='black', linewidth=1, label='Training R
    plt.plot(epochs, val_recall, color='red', linewidth=1, label='Validation Rec

    plt.xlabel('Epochs')
    plt.ylabel('Recall')
    plt.title('Training and Validation Recall Over Time')
    plt.grid(True, linestyle='--', alpha=0.7)
    plt.legend()
    plt.gca().set_facecolor('#f8f9fa')

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

```
def extract_metrics(history):
    metrics = {}
    for metric in ['loss', 'val_loss', 'recall', 'val_recall', 'precision', 'val
        if metric in history.history:
            metrics[metric] = history.history[metric]
        else:
            print(f"Warning: {metric} not found in history")
            metrics[metric] = None
    return metrics
```

In [4]:
```
# importing data
image_dir = "/content/drive/MyDrive/Antonio Pilan/projeto_final_introML/Images"
image_tensor_rgb, label_tensor = load_and_preprocess_images(image_dir, grayscale
label_tensor = transform_labels(label_tensor) #transform label tensor in boolean

X_train, X_test, y_train, y_test = split_data(image_tensor_rgb, label_tensor, te

#separating each pixel tensor in a separate variable
red_channel = X_train[:, :, :, 0]
green_channel = X_train[:, :, :, 1]
blue_channel = X_train[:, :, :, 2]

#transform rgb tensors into grayscale
image_tensor = tf.image.rgb_to_grayscale(X_train)
grayscale_channel = image_tensor

print("Shape of red channel tensor:", red_channel.shape)
print("Shape of green channel tensor:", green_channel.shape)
print("Shape of blue channel tensor:", blue_channel.shape)
print("Shape of grayscale channel tensor:", grayscale_channel.shape)
print("Shape of Grayscale image tensor:", image_tensor.shape)

print("Shape of label tensor (training):", y_train.shape)
print("Shape of label tensor (test):", y_test.shape)


print("Shape of label tensor:", label_tensor.shape)
```

```
Shape of red channel tensor: (2205, 224, 224)
Shape of green channel tensor: (2205, 224, 224)
Shape of blue channel tensor: (2205, 224, 224)
Shape of grayscale channel tensor: (2205, 224, 224, 1)
Shape of Grayscale image tensor: (2205, 224, 224, 1)
Shape of label tensor (training): (2205,)
Shape of label tensor (test): (3308,)
Shape of label tensor: (5513,)
```

In [5]:
```
counts = np.bincount(y_train)
total = len(y_train)
print(
    "Number of positive samples in training data: {} ({:.2f}% of total)".format(
        counts[1], 100 * float(counts[1]) / len(y_train)
    )
)

weight_for_0 = total/counts[0]
weight_for_1 = total/counts[1]

print("Weight for class 0: {:.2f}".format(weight_for_0))
```

```
print("Weight for class 1: {:.2f}".format(weight_for_1))
##############################################################################
counts = np.bincount(y_test)
total = len(y_test)
print(
    "Number of positive samples in test data: {} ({:.2f}% of total)".format(
        counts[1], 100 * float(counts[1]) / len(y_train)
    )
)
```

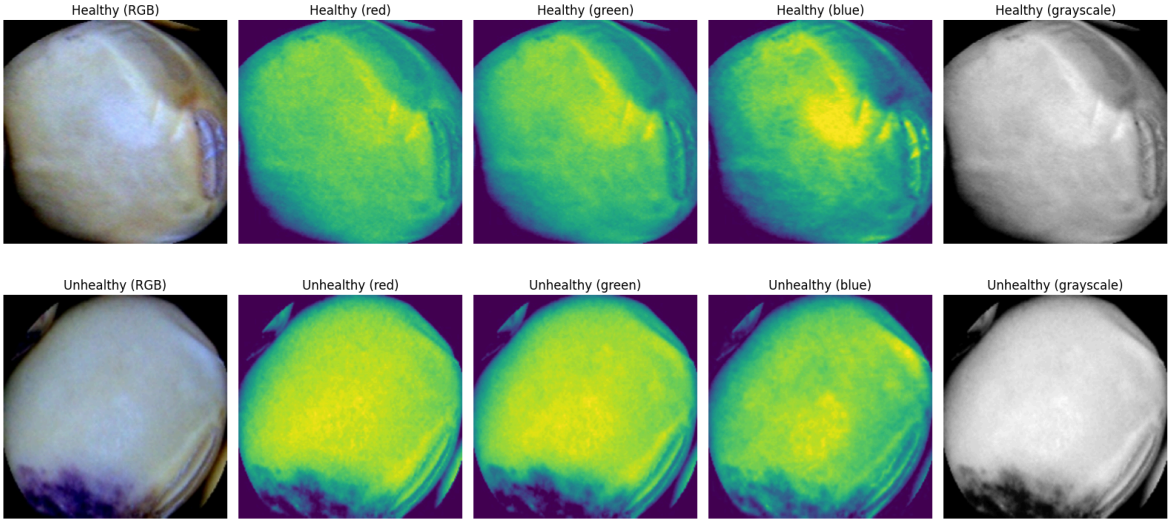
```
Number of positive samples in training data: 480 (21.77% of total)
Weight for class 0: 1.28
Weight for class 1: 4.59
Number of positive samples in test data: 721 (32.70% of total)
```

In [6]:

```
# Find first healthy and unhealthy seed indices
healthy_idx = None
unhealthy_idx = None

for i in range(len(y_train)):
    if y_train[i] == 1 and healthy_idx is None:
        healthy_idx = i
    elif y_train[i] == 0 and unhealthy_idx is None:
        unhealthy_idx = i

    # Break if we found both
    if healthy_idx is not None and unhealthy_idx is not None:
        break

print(f"First healthy seed index: {healthy_idx}")
print(f"First unhealthy seed index: {unhealthy_idx}")

channels = {
    'red': red_channel,
    'green': green_channel,
    'blue': blue_channel,
    'grayscale': grayscale_channel
}

# Plot the channels if indices were found
if healthy_idx is not None and unhealthy_idx is not None:
    plt.figure(figsize=(16, 8))

    # Plot the full RGB images
    plt.subplot(2, 5, 1)
    plt.imshow(X_train[healthy_idx])
    plt.title("Healthy (RGB)")
    plt.axis('off')

    plt.subplot(2, 5, 6)
    plt.imshow(X_train[unhealthy_idx])
    plt.title("Unhealthy (RGB)")
    plt.axis('off')

    # Plot each channel
    for i, (channel_name, channel_data) in enumerate(channels.items(), 1):
        if channel_name == 'grayscale':
            healthy_img = tf.squeeze(channel_data[healthy_idx]).numpy()
            unhealthy_img = tf.squeeze(channel_data[unhealthy_idx]).numpy()
            cmap_val = 'gray'
```

```
        else:
            healthy_img = channel_data[healthy_idx]
            unhealthy_img = channel_data[unhealthy_idx]
            cmap_val = 'viridis'

        plt.subplot(2, 5, i+1)
        plt.imshow(healthy_img, cmap=cmap_val)
        plt.title(f"Healthy ({channel_name})")
        plt.axis('off')

        plt.subplot(2, 5, i+6)
        plt.imshow(unhealthy_img, cmap=cmap_val)
        plt.title(f"Unhealthy ({channel_name})")
        plt.axis('off')

    plt.tight_layout()
    plt.show()
else:
    print("Could not find both healthy and unhealthy examples.")
```

```
First healthy seed index: 8
First unhealthy seed index: 0
```



Dealing with class imbalance

Model Simulations

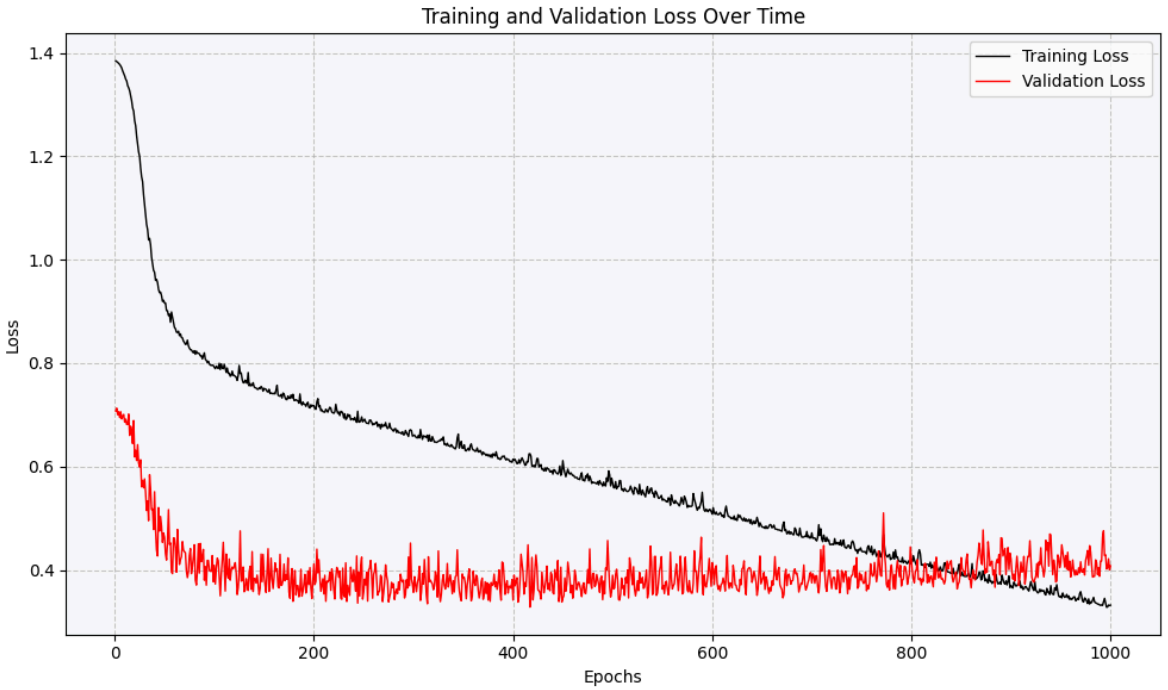
First exploration: Convolutional Neural Networks with Greyscale images

In [7]:
```
X_train_grayscale, X_val_grayscale, y_train_grayscale, y_val_grayscale = split_d
y_train_grayscale = y_train_grayscale.reshape(-1, 1)
y_val_grayscale = y_val_grayscale.reshape(-1, 1)

model_convolutional = tf.keras.models.Sequential([
    Conv2D(128, (3, 3), activation='relu', input_shape=(224, 224, 1)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
```

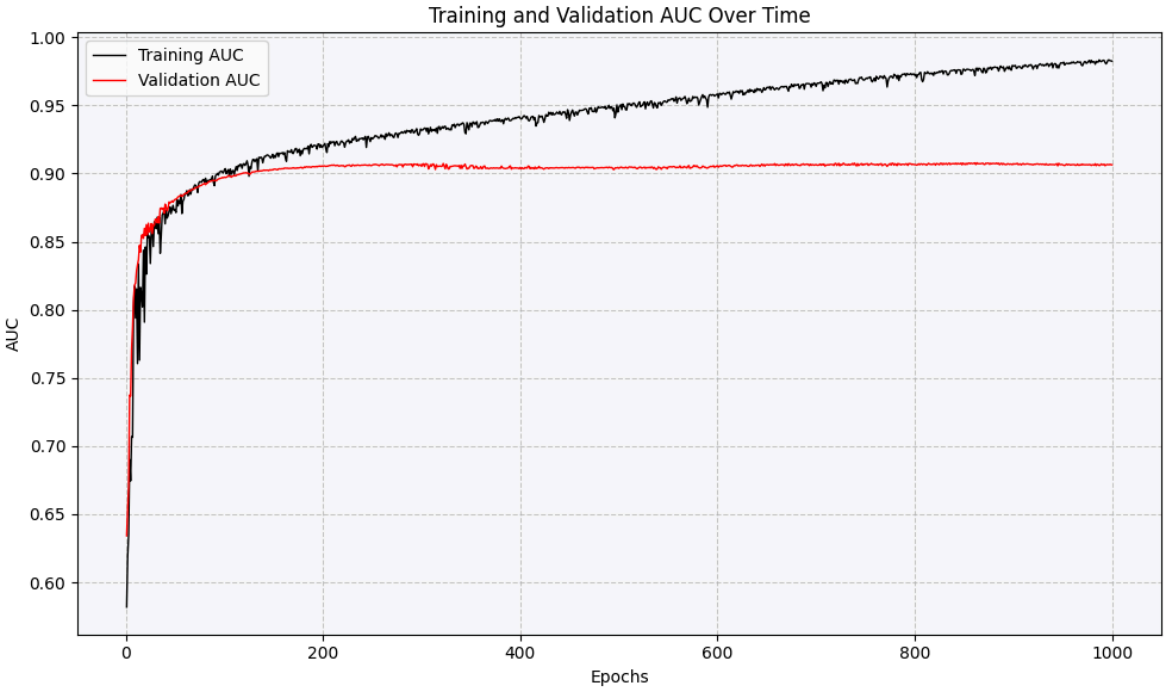
```
    Conv2D(32, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(32, activation='relu'),
    Dense(1, activation='sigmoid') # Output layer for binary classification
])

metrics = [
    keras.metrics.F1Score(name="f1score"),
    keras.metrics.Accuracy(name="accuracy"),
    keras.metrics.Precision(name="precision"),
    keras.metrics.Recall(name="recall"),
    keras.metrics.AUC(name="AUC"),
]

model_convolutional.compile(optimizer=keras.optimizers.Adam(learning_rate=0.0000
                            loss='binary_crossentropy', metrics=metrics)

# Train the model
class_weight = {0: weight_for_0, 1: weight_for_1}
history_convolutional = model_convolutional.fit(
                                                X_train_grayscale, y_train_grays
                                                epochs=1000, batch_size=128,
                                                validation_data=(X_val_grayscale
                                                class_weight=class_weight,
                                                verbose=0
                                                )
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base_conv.
py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a laye
r. When using Sequential models, prefer using an `Input(shape)` object as the fir
st layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

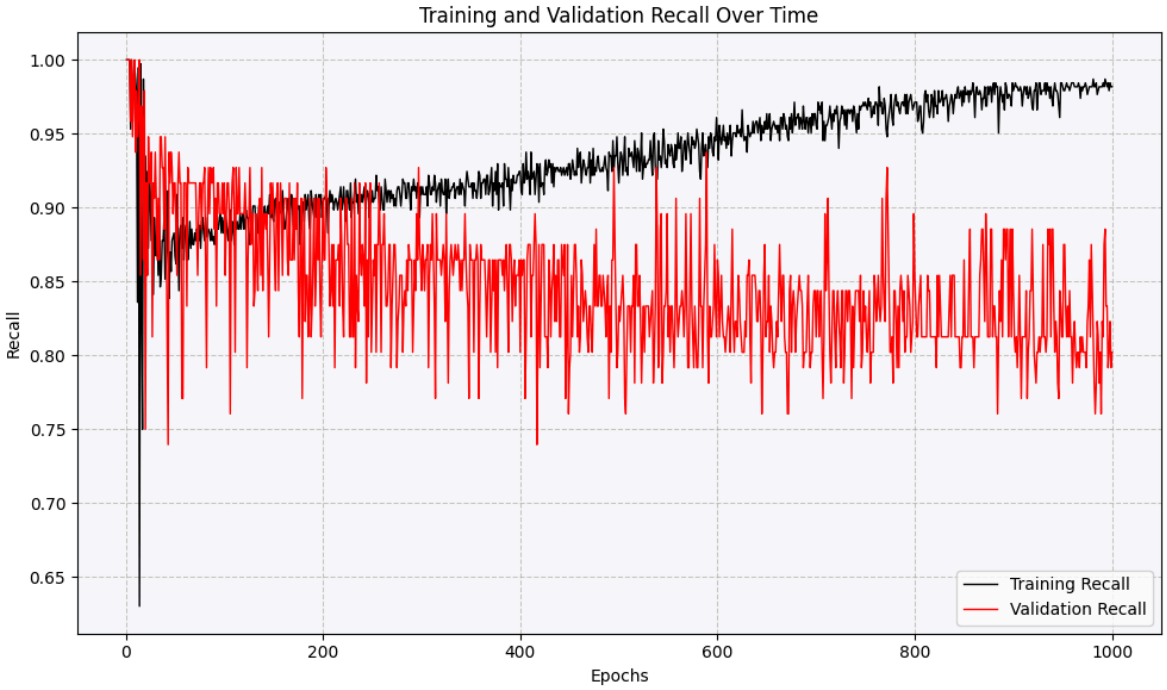
In [8]:
```
plot_learning_process(history_convolutional)
```



In [9]:
```
plot_auc_metrics(history_convolutional)
```



In [10]:
```
plot_recall_metrics(history_convolutional)
```

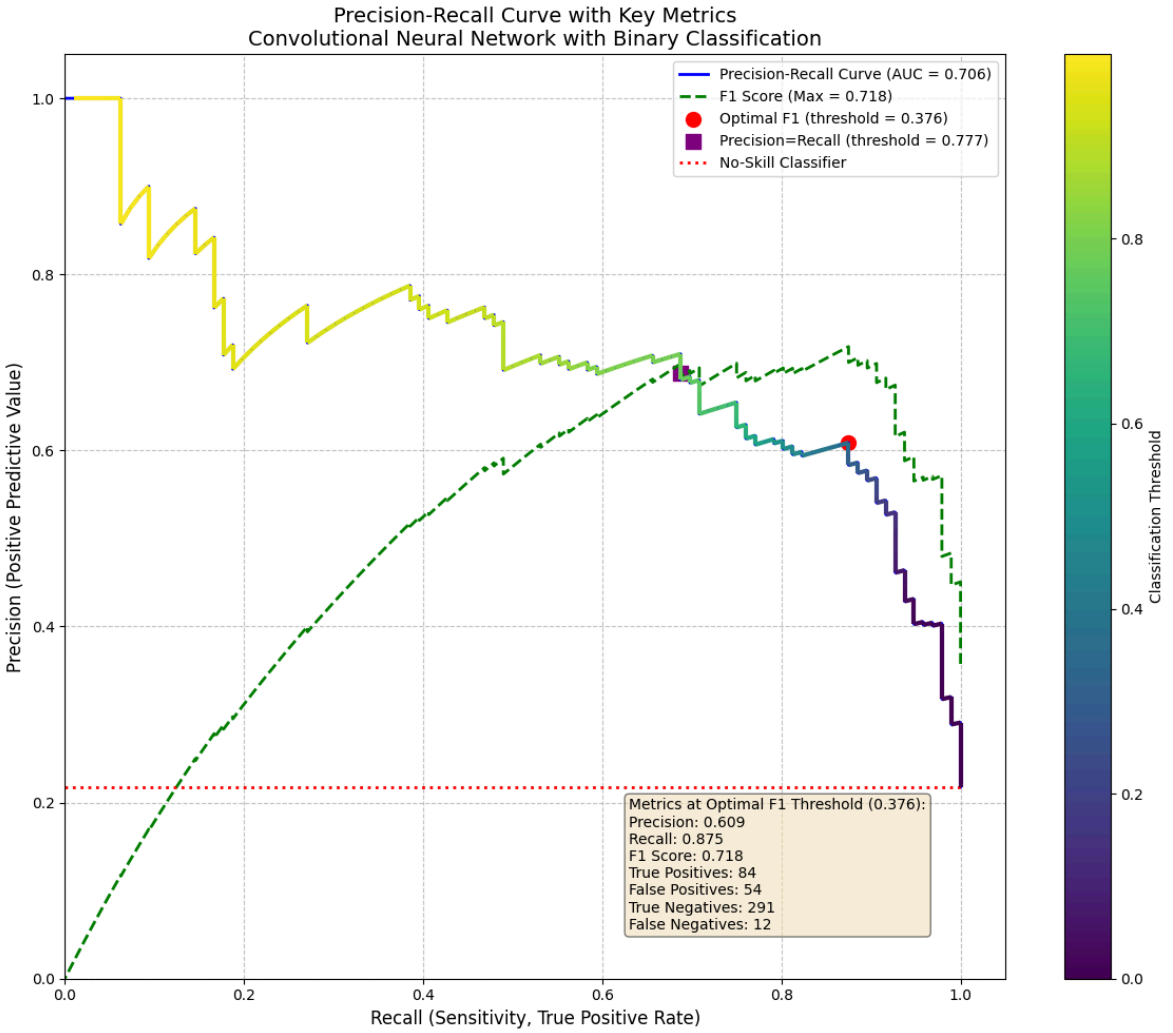


In [11]:
```
y_pred_proba = model_convolutional.predict(X_val_grayscale)
precisions, recalls, thresholds = precision_recall_curve(y_val_grayscale, y_pred
f1_scores = 2 * (precisions * recalls) / (precisions + recalls)
optimal_idx = np.argmax(f1_scores)
optimal_threshold_f1_score = thresholds[optimal_idx]

average_precision = average_precision_score(y_val_grayscale, y_pred_proba)

pr_auc = auc(recalls, precisions)

# Find optimal thresholds for different metrics
pr_diff = np.abs(precisions[:-1] - recalls[:-1])
optimal_pr_idx = np.argmin(pr_diff)
optimal_threshold_pr_equal = thresholds[optimal_pr_idx]
```

```
plt.figure(figsize=(12, 10))
plt.plot(recalls, precisions, color='b', lw=2, label=f'Precision-Recall Curve (A
plt.plot(recalls, f1_scores, color='green', lw=2, linestyle='--',
         label=f'F1 Score (Max = {f1_scores[optimal_idx]:.3f})')
plt.scatter(recalls[optimal_idx], precisions[optimal_idx], marker='o', color='re
            label=f'Optimal F1 (threshold = {optimal_threshold_f1_score:.3f})')
plt.scatter(recalls[optimal_pr_idx], precisions[optimal_pr_idx], marker='s', col
            label=f'Precision=Recall (threshold = {optimal_threshold_pr_equal:.3

# Calculate and display metrics at the optimal F1 threshold
y_pred_optimal = (y_pred_proba >= optimal_threshold_f1_score).astype(int)
tn, fp, fn, tp = confusion_matrix(y_val_grayscale, y_pred_optimal).ravel()
precision_optimal = tp / (tp + fp) if (tp + fp) > 0 else 0
recall_optimal = tp / (tp + fn) if (tp + fn) > 0 else 0
f1_optimal = 2 * (precision_optimal * recall_optimal) / (precision_optimal + rec

# Textbox
textbox_props = dict(boxstyle='round', facecolor='wheat', alpha=0.5)
textbox_content = '\n'.join((
    f'Metrics at Optimal F1 Threshold ({optimal_threshold_f1_score:.3f}):',
    f'Precision: {precision_optimal:.3f}',
    f'Recall: {recall_optimal:.3f}',
    f'F1 Score: {f1_optimal:.3f}',
    f'True Positives: {tp}',
    f'False Positives: {fp}',
    f'True Negatives: {tn}',
    f'False Negatives: {fn}'
))
plt.text(0.6, 0.05, textbox_content, transform=plt.gca().transAxes,
         fontsize=10, verticalalignment='bottom', bbox=textbox_props)

# No-skill classifier line
plt.plot([0, 1], [np.sum(y_val_grayscale) / len(y_val_grayscale)] * 2,
         color='red', linestyle=':', lw=2, label='No-Skill Classifier')

# Personalization
plt.title('Precision-Recall Curve with Key Metrics\nConvolutional Neural Network
plt.xlabel('Recall (Sensitivity, True Positive Rate)', fontsize=12)
plt.ylabel('Precision (Positive Predictive Value)', fontsize=12)

plt.xlim([0.0, 1.05])
plt.ylim([0.0, 1.05])

plt.grid(True, linestyle='--', alpha=0.7)
plt.legend(loc='upper right', fontsize=10)

# Add visualization of threshold changes along the curve
from matplotlib.collections import LineCollection
from matplotlib.colors import ListedColormap, BoundaryNorm

points = np.array([recalls[:-1], precisions[:-1]]).T.reshape(-1, 1, 2)
segments = np.concatenate([points[:-1], points[1:]], axis=1)

norm = plt.Normalize(thresholds.min(), thresholds.max())
lc = LineCollection(segments, cmap='viridis', norm=norm)
lc.set_array(thresholds)
lc.set_linewidth(3)

line = plt.gca().add_collection(lc)
```

```
cbar = plt.colorbar(line, ax=plt.gca())
cbar.set_label('Classification Threshold', fontsize=10)
plt.tight_layout()

plt.show()

# Additional metrics for analysis
print(f"Average Precision (AP): {average_precision:.4f}")
print(f"Area Under PR Curve: {pr_auc:.4f}")
print(f"Optimal F1 Score Threshold: {optimal_threshold_f1_score:.4f} (F1 = {f1_s
print(f"Precision=Recall Threshold: {optimal_threshold_pr_equal:.4f} (Value = {p
```

```
14/14 ━━━━━━━━━━━━━━━━━━━━ 1s 51ms/step
```




```
Average Precision (AP): 0.7089
Area Under PR Curve: 0.7064
Optimal F1 Score Threshold: 0.3764 (F1 = 0.7179)
Precision=Recall Threshold: 0.7767 (Value = 0.6875)
```

Now, we'll go towards more cientific aproach: K-fold validation and sepparating the test set to avoid *information leaks*

Performance metrics: k-fold overfitting analysis and redesigning our model for grayscale

In [12]:
```
# model as a function
def create_model(X_train, y_train, X_test=None, y_test=None, class_weight=None,
    model_convolutional = tf.keras.models.Sequential([
```

```
      Conv2D(128, (3, 3), activation='relu', input_shape=(224, 224, 1)),
      MaxPooling2D((2, 2)),
      Conv2D(64, (3, 3), activation='relu'),
      MaxPooling2D((2, 2)),
      Conv2D(32, (3, 3), activation='relu'),
      MaxPooling2D((2, 2)),
      Flatten(),
      Dense(32, activation='relu'),
      Dense(1, activation='sigmoid') # Output layer for binary classification
      ])

    metrics = [
        keras.metrics.F1Score(name="f1score"),
        keras.metrics.Precision(name="precision"),
        keras.metrics.Recall(name="recall"),
        keras.metrics.Accuracy(name="accuracy")
    ]

    model_convolutional.compile(optimizer=keras.optimizers.Adam(learning_rate=0.
                                loss='binary_crossentropy', metrics=metrics)

    # Train the model
    class_weight = {0: weight_for_0, 1: weight_for_1}
    if X_test is not None and y_test is not None:
      print("Simulation model")
      history_convolutional = model_convolutional.fit(
                                                      X_train, y_train,
                                                      epochs=epochs, batch_size=
                                                      validation_data=(X_test, y
                                                      class_weight=class_weight,
                                                      verbose = 0
                                                      )
    else:
      print("Final model")
      history_convolutional = model_convolutional.fit(
                                                      X_train, y_train,
                                                      epochs=epochs, batch_size=
                                                      class_weight=class_weight,
                                                      verbose = 0
                                                      )

    return model_convolutional, history_convolutional
```

The following validation cell was generated via Claude:

In [13]:
```
# Initialize variables to store best model and metrics
best_model = None
best_fold_metrics = None
best_f1_score = -1
best_fold_idx = -1
best_threshold = 0.5
best_class_weight = None

# Storage for metrics across all folds
all_fold_metrics = {}
f1_scores_across_folds = []
pr_aucs_across_folds = []
roc_aucs_across_folds = []
thresholds_across_folds = []
```

```
minimal_epoch = []

# Storage for curves data for averaging
all_precisions = []
all_recalls = []
mean_recall = np.linspace(0, 1, 100)  # For interpolation of PR curves
all_tprs = []
all_fprs = []
mean_fpr = np.linspace(0, 1, 100)  # For interpolation of ROC curves

# Set up figure for visualization
plt.figure(figsize=(20, 10))
plt.subplot(1, 2, 1)  # For Precision-Recall curve
plt.subplot(1, 2, 2)  # For ROC curve

# Prepare your data
X_data = grayscale_channel  # Your input data (assuming grayscale_channel is def
y_data = y_train  # Your target variable (assuming y_train is defined)

# Start k-fold cross-validation
for k in range(5):

    # Split data for this fold
    X_train_fold, X_val_fold, y_train_fold, y_val_fold = split_data(X_data, y_da
    y_train_fold = y_train_fold.reshape(-1, 1)
    y_val_fold = y_val_fold.reshape(-1, 1)

    # Calculate class weights for this fold
    counts = np.bincount(y_train_fold.flatten())
    total = len(y_train_fold)
    print(
        "Number of positive samples in training data: {} ({:.2f}% of total)".for
            counts[1], 100 * float(counts[1]) / len(y_train)
        )
    )

    weight_for_0 = total/counts[0]
    weight_for_1 = total/counts[1]
    class_weight = {0: weight_for_0, 1: weight_for_1}

    print(f"Class weights - Class 0: {weight_for_0:.3f}, Class 1: {weight_for_1:

    # Use your create_model function
    fold_model, fold_history = create_model(
        X_train_fold, y_train_fold,
        X_val_fold, y_val_fold,
        class_weight
    )

    # Get predictions on validation set
    y_pred_proba = fold_model.predict(X_val_fold)

    # Ensure we have the correct shape for y_pred_proba
    if len(y_pred_proba.shape) > 1 and y_pred_proba.shape[1] > 1:
        y_pred_proba = y_pred_proba[:, 1]

    # ===== F1 Score and PR Curve Analysis =====
    # Calculate precision-recall curve and metrics
    precisions, recalls, thresholds = precision_recall_curve(y_val_fold, y_pred_
```

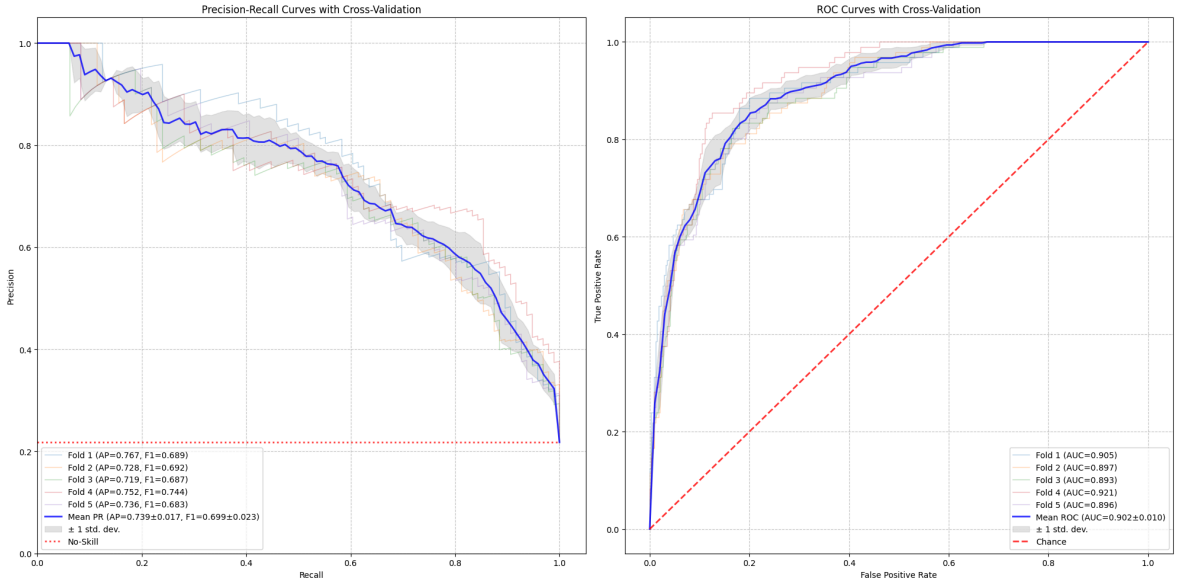
```
    # Calculate F1 scores for each threshold
    # Add small epsilon to avoid division by zero
    epsilon = 1e-8
    f1_scores = 2 * (precisions * recalls) / (precisions + recalls + epsilon)

    # Find optimal threshold that maximizes F1 score
    optimal_idx = np.argmax(f1_scores[:-1])  # Exclude the last point as it does
    optimal_threshold_f1 = thresholds[optimal_idx]
    max_f1_score = f1_scores[optimal_idx]

    # Calculate average precision and PR AUC
    average_precision = average_precision_score(y_val_fold, y_pred_proba)
    pr_auc = auc(recalls, precisions)

    # Find point where precision equals recall
    pr_diff = np.abs(precisions[:-1] - recalls[:-1])
    optimal_pr_idx = np.argmin(pr_diff)
    optimal_threshold_pr_equal = thresholds[optimal_pr_idx]

    # ===== ROC Curve Analysis =====
    # Calculate ROC curve and AUC
    fpr, tpr, roc_thresholds = roc_curve(y_val_fold, y_pred_proba)
    roc_auc = auc(fpr, tpr)

    # Interpolate the TPR values for ROC curve averaging
    interp_tpr = np.interp(mean_fpr, fpr, tpr)
    interp_tpr[0] = 0.0  # Force the curve to start at (0,0)
    all_tprs.append(interp_tpr)

    # Interpolate precision values for PR curve averaging
    # Note: PR curve is not as easily averaged as ROC due to its non-monotonic n
    interp_precision = np.interp(mean_recall, recalls[::-1], precisions[::-1])
    all_precisions.append(interp_precision)

    # ===== Calculate metrics at optimal threshold =====
    y_pred_optimal = (y_pred_proba >= optimal_threshold_f1).astype(int)
    tn, fp, fn, tp = confusion_matrix(y_val_fold, y_pred_optimal).ravel()

    precision_optimal = tp / (tp + fp) if (tp + fp) > 0 else 0
    recall_optimal = tp / (tp + fn) if (tp + fn) > 0 else 0
    f1_optimal = 2 * (precision_optimal * recall_optimal) / (precision_optimal +

    # ===== Store metrics for this fold =====
    fold_metrics = {
        "model": fold_model,
        "history": fold_history,
        "val_loss": np.min(fold_history.history['val_loss']),
        "epoch_min_val_loss": np.argmin(fold_history.history['val_loss']),
        "f1_optimal": f1_optimal,
        "precision_optimal": precision_optimal,
        "recall_optimal": recall_optimal,
        "pr_auc": pr_auc,
        "average_precision": average_precision,
        "roc_auc": roc_auc,
        "optimal_threshold_f1": optimal_threshold_f1,
        "optimal_threshold_pr_equal": optimal_threshold_pr_equal,
        "confusion_matrix": {
            "true_negatives": tn,
            "false_positives": fp,
            "false_negatives": fn,
```

```
            "true_positives": tp
        },
        "class_weight": class_weight
    }

    all_fold_metrics[k] = fold_metrics
    f1_scores_across_folds.append(f1_optimal)
    pr_aucs_across_folds.append(pr_auc)
    roc_aucs_across_folds.append(roc_auc)
    thresholds_across_folds.append(optimal_threshold_f1)
    minimal_epoch.append(np.argmin(fold_history.history['val_loss']))

    # Check if this is the best fold so far based on F1 score
    if f1_optimal > best_f1_score:
        best_f1_score = f1_optimal
        best_fold_idx = k
        best_model = fold_model
        best_fold_metrics = fold_metrics
        best_threshold = optimal_threshold_f1
        best_class_weight = class_weight

    # Print fold results
    print(f"\nFold {k+1} Results:")
    print(f"F1 Score: {f1_optimal:.4f} at threshold {optimal_threshold_f1:.4f}")
    print(f"Precision: {precision_optimal:.4f}, Recall: {recall_optimal:.4f}")
    print(f"PR AUC: {pr_auc:.4f}, ROC AUC: {roc_auc:.4f}")
    print(f"Confusion Matrix: TN={tn}, FP={fp}, FN={fn}, TP={tp}")

    # Plot PR curve for this fold
    plt.subplot(1, 2, 1)
    plt.plot(recalls, precisions, lw=1, alpha=0.3,
             label=f'Fold {k+1} (AP={average_precision:.3f}, F1={f1_optimal:.3f}

    # Plot ROC curve for this fold
    plt.subplot(1, 2, 2)
    plt.plot(fpr, tpr, lw=1, alpha=0.3,
             label=f'Fold {k+1} (AUC={roc_auc:.3f})')

# Calculate mean metrics across folds
mean_f1 = np.mean(f1_scores_across_folds)
std_f1 = np.std(f1_scores_across_folds)
mean_pr_auc = np.mean(pr_aucs_across_folds)
std_pr_auc = np.std(pr_aucs_across_folds)
mean_roc_auc = np.mean(roc_aucs_across_folds)
std_roc_auc = np.std(roc_aucs_across_folds)
mean_threshold = np.mean(thresholds_across_folds)

# Calculate mean PR curve
mean_precision = np.mean(all_precisions, axis=0)
std_precision = np.std(all_precisions, axis=0)

# Calculate mean ROC curve
mean_tpr = np.mean(all_tprs, axis=0)
mean_tpr[-1] = 1.0  # Force the curve to end at (1,1)
std_tpr = np.std(all_tprs, axis=0)
tprs_upper = np.minimum(mean_tpr + std_tpr, 1)
tprs_lower = np.maximum(mean_tpr - std_tpr, 0)

# Finalize PR Curve plot
plt.subplot(1, 2, 1)
```

```
plt.plot(mean_recall, mean_precision, color='b',
         label=f'Mean PR (AP={mean_pr_auc:.3f}±{std_pr_auc:.3f}, F1={mean_f1:.3f
         lw=2, alpha=0.8)
# Add confidence interval for PR curve (this is approximate due to PR curve's na
plt.fill_between(mean_recall,
                 np.maximum(mean_precision - std_precision, 0),
                 np.minimum(mean_precision + std_precision, 1),
                 color='grey', alpha=0.2,
                 label='± 1 std. dev.')
# Plot baseline (no skill classifier)
plt.plot([0, 1], [np.mean(y_data)] * 2, linestyle=':', color='r',
         label='No-Skill', lw=2, alpha=0.8)

plt.xlim([0.0, 1.05])
plt.ylim([0.0, 1.05])
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curves with Cross-Validation')
plt.legend(loc='lower left')
plt.grid(True, linestyle='--', alpha=0.7)

# Finalize ROC Curve plot
plt.subplot(1, 2, 2)
plt.plot(mean_fpr, mean_tpr, color='b',
         label=f'Mean ROC (AUC={mean_roc_auc:.3f}±{std_roc_auc:.3f})',
         lw=2, alpha=0.8)
plt.fill_between(mean_fpr, tprs_lower, tprs_upper, color='grey', alpha=0.2,
                 label='± 1 std. dev.')
plt.plot([0, 1], [0, 1], linestyle='--', lw=2, color='r',
         label='Chance', alpha=.8)
plt.xlim([-0.05, 1.05])
plt.ylim([-0.05, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curves with Cross-Validation')
plt.legend(loc='lower right')
plt.grid(True, linestyle='--', alpha=0.7)

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

```
Number of positive samples in training data: 384 (17.41% of total)
Class weights - Class 0: 1.278, Class 1: 4.594
/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base_conv.
py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a laye
r. When using Sequential models, prefer using an `Input(shape)` object as the fir
st layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Simulation model
14/14 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step

Fold 1 Results:
F1 Score: 0.6888 at threshold 0.6493
Precision: 0.5724, Recall: 0.8646
PR AUC: 0.7658, ROC AUC: 0.9049
Confusion Matrix: TN=283, FP=62, FN=13, TP=83
Number of positive samples in training data: 384 (17.41% of total)
Class weights - Class 0: 1.278, Class 1: 4.594
Simulation model
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base_conv.
py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a laye
r. When using Sequential models, prefer using an `Input(shape)` object as the fir
st layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
14/14 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step

Fold 2 Results:
F1 Score: 0.6923 at threshold 0.6837
Precision: 0.7326, Recall: 0.6562
PR AUC: 0.7262, ROC AUC: 0.8974
Confusion Matrix: TN=322, FP=23, FN=33, TP=63
Number of positive samples in training data: 384 (17.41% of total)
Class weights - Class 0: 1.278, Class 1: 4.594
Simulation model
/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base_conv.
py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a laye
r. When using Sequential models, prefer using an `Input(shape)` object as the fir
st layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
14/14 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step

Fold 3 Results:
F1 Score: 0.6866 at threshold 0.7285
Precision: 0.6571, Recall: 0.7188
PR AUC: 0.7169, ROC AUC: 0.8933
Confusion Matrix: TN=309, FP=36, FN=27, TP=69
Number of positive samples in training data: 384 (17.41% of total)
Class weights - Class 0: 1.278, Class 1: 4.594
Simulation model
/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base_conv.
py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a laye
r. When using Sequential models, prefer using an `Input(shape)` object as the fir
st layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
14/14 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step

Fold 4 Results:
F1 Score: 0.7442 at threshold 0.5802
Precision: 0.6723, Recall: 0.8333
PR AUC: 0.7502, ROC AUC: 0.9208
Confusion Matrix: TN=306, FP=39, FN=16, TP=80
Number of positive samples in training data: 384 (17.41% of total)
Class weights - Class 0: 1.278, Class 1: 4.594
Simulation model
/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base_conv.
py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a laye
r. When using Sequential models, prefer using an `Input(shape)` object as the fir
st layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
14/14 ━━━━━━━━━━━━━━━━━━━━ 1s 23ms/step

Fold 5 Results:
F1 Score: 0.6827 at threshold 0.7062
Precision: 0.6339, Recall: 0.7396
PR AUC: 0.7346, ROC AUC: 0.8955
Confusion Matrix: TN=304, FP=41, FN=25, TP=71
```



In [14]:
```
# Print summary of cross-validation results
print("\n" + "="*80)
print("Cross Validation Results:")
print("="*80)
print(f"Mean F1 Score: {mean_f1:.4f} ± {std_f1:.4f}")
print(f"Mean PR AUC: {mean_pr_auc:.4f} ± {std_pr_auc:.4f}")
print(f"Mean ROC AUC: {mean_roc_auc:.4f} ± {std_roc_auc:.4f}")
print(f"Mean Optimal Threshold: {mean_threshold:.4f}")
print("\nIndividual Fold Metrics:")
metrics_df = pd.DataFrame({
    'Fold': range(1, 6),
    'F1 Score': f1_scores_across_folds,
    'PR AUC': pr_aucs_across_folds,
    'ROC AUC': roc_aucs_across_folds,
    'Optimal Threshold': thresholds_across_folds,
    'minimal epoch': minimal_epoch,
})
print(metrics_df)
```

```
================================================================================
Cross Validation Results:
================================================================================
Mean F1 Score: 0.6989 ± 0.0229
Mean PR AUC: 0.7387 ± 0.0174
Mean ROC AUC: 0.9024 ± 0.0100
Mean Optimal Threshold: 0.6696

Individual Fold Metrics:
   Fold  F1 Score    PR AUC   ROC AUC  Optimal Threshold  minimal epoch
0     1  0.688797  0.765762  0.904921           0.649347            174
1     2  0.692308  0.726171  0.897373           0.683708            188
2     3  0.686567  0.716947  0.893267           0.728520            184
3     4  0.744186  0.750193  0.920773           0.580238            185
4     5  0.682692  0.734560  0.895501           0.706202            197
```

Developing the final model

In [15]:
```
y_train_final = y_train.reshape(-1, 1)

# final model implementation
```

```
counts = np.bincount(y_train.flatten())
total = len(y_train)
print(
    "Number of positive samples in training data: {} ({:.2f}% of total)".format(
        counts[1], 100 * float(counts[1]) / len(y_train)
    )
)

weight_for_0 = total/counts[0]
weight_for_1 = total/counts[1]
class_weight = {0: weight_for_0, 1: weight_for_1}

epochs = int(metrics_df['minimal epoch'].mean()) #that minimizes loss
optimal_treshold = metrics_df['Optimal Threshold'].mean() #that maximizes f1-sco

print(f"Class weights - Class 0: {weight_for_0:.3f}, Class 1: {weight_for_1:.3f}

final_model = tf.keras.models.Sequential([
    Conv2D(128, (3, 3), activation='relu', input_shape=(224, 224, 1)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(32, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(32, activation='relu'),
    Dense(1, activation='sigmoid') # Output layer for binary classification
])

metrics = [
    keras.metrics.F1Score(name="f1score"),
    keras.metrics.Accuracy(name="accuracy"),
    keras.metrics.Precision(name="precision"),
    keras.metrics.Recall(name="recall"),
    keras.metrics.AUC(name="AUC"),
]

final_model.compile(optimizer=keras.optimizers.Adam(learning_rate=0.00001),
                            loss='binary_crossentropy', metrics=metrics)

# Train the model
history = final_model.fit(
                                    grayscale_channel, y_train_final,
                                    epochs=epochs, batch_size=128,
                                    class_weight=class_weight,
                                    verbose=0
                                    )
```

```
Number of positive samples in training data: 480 (21.77% of total)
Class weights - Class 0: 1.278, Class 1: 4.594
```

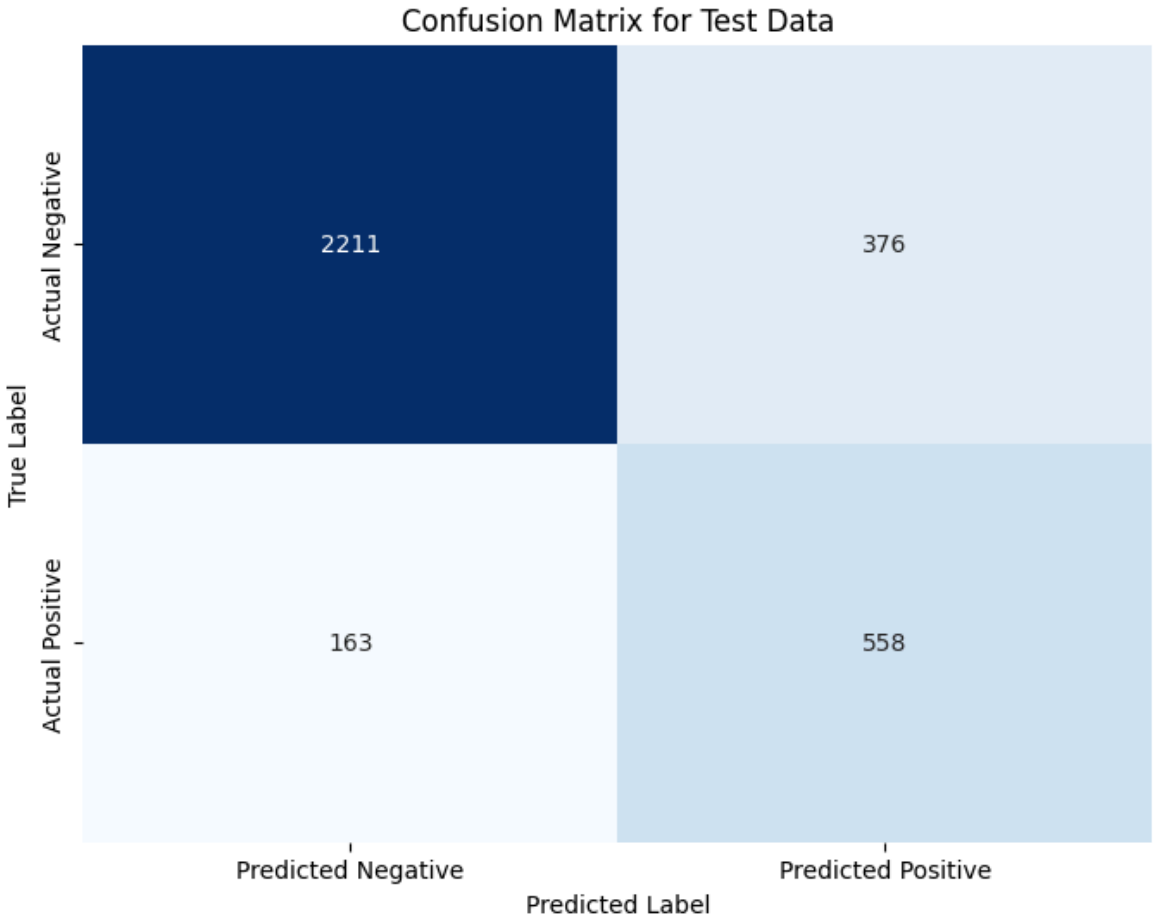
```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base_conv.
py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a laye
r. When using Sequential models, prefer using an `Input(shape)` object as the fir
st layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

In [16]:

```
X_test_grayscale = tf.image.rgb_to_grayscale(X_test)
y_pred_proba_test = final_model.predict(X_test_grayscale)

# Convert probabilities to binary predictions
```

```
y_pred_test = (y_pred_proba_test >= optimal_treshold).astype(int)

# Calculate and print test metrics
test_accuracy = accuracy_score(y_test, y_pred_test)
test_precision = precision_score(y_test, y_pred_test)
test_recall = recall_score(y_test, y_pred_test)
test_f1_score = f1_score(y_test, y_pred_test)
test_confusion_matrix = confusion_matrix(y_test, y_pred_test)

print("\n" + "="*80)
print("Final Model performance (TEST DATA)")
print("="*80)
print(f"Optimal Threshold Used (from CV mean): {optimal_treshold:.4f}")
print(f"Test Accuracy: {test_accuracy:.4f}")
print(f"Test Precision: {test_precision:.4f}")
print(f"Test Recall: {test_recall:.4f}")
print(f"Test F1 Score: {test_f1_score:.4f}")

# Plot the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(test_confusion_matrix, annot=True, fmt='d', cmap='Blues', cbar=False
            xticklabels=['Predicted Negative', 'Predicted Positive'],
            yticklabels=['Actual Negative', 'Actual Positive'])
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix for Test Data')
plt.show()
```

```
104/104 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step

================================================================================
Final Model performance (TEST DATA)
================================================================================
Optimal Threshold Used (from CV mean): 0.6696
Test Accuracy: 0.8371
Test Precision: 0.5974
Test Recall: 0.7739
Test F1 Score: 0.6743
```


Confusion Matrix for Test Data
Actual Negative
Actual Positive
True Label
2211
376
163
558
Predicted Negative
Predicted Positive
Predicted Label