Automated Image Detection of LeBron James with Deep Learning

William Pham Joy Zhang Jacob Quisumbing Andrew Song Twesha Ghosh

May 15, 2025

Contents

1 Introduction

1.1 Problem Statement

This project aims to develop a deep learning-based model capable of detecting whether LeBron James is present in an image. In later stages, we aim to extend this capability to video detection, allowing real-time or batch processing of video footage to identify frames containing LeBron James. The ability to identify LeBron James in various media formats can be useful for sports analysis, highlight generation, and fan engagement applications.

1.2 Motivation

The ability to automatically identify individuals in visual media has significant applications across sports analytics, media production, and fan engagement platforms. LeBron James, as one of the most prominent and widely recognized figures in professional basketball, presents an ideal starting point for developing and testing such capabilities for our project. Automating the detection of LeBron James in images and video can greatly enhance the efficiency of tasks, such as highlight generation, player tracking, and content indexing, which are currently time-consuming and often performed manually.

This project aims to explore the effectiveness of deep learning techniques, specifically Convolutional Neural Networks (CNNs), to distinguish LeBron James from other individuals in a wide variety of visual contexts, including game footage, interviews, and social media imagery. By using a targeted approach focused on a single athlete, we aim to build a robust foundation that can later be generalized to support multi-player detection and advanced video analysis.

Our project will ultimately contribute to the growing field of automated sports analytics and demonstrates the potential of computer vision technologies to transform the way visual sports data is processed and utilized.

2 Data Collection

2.1 Data Source

For data collection, we curated a diverse set of image datasets from Roboflow Universe, selecting a total of five publicly available collections. One dataset consisted mainly of images of LeBron James, which we used as our positive classification set. The remaining four datasets included images of various NBA players and were used for the negative classification set to help the model learn to distinguish LeBron James from other individuals commonly found in basketball-related media.

2.2 Data Preprocessing

In total, our negative classification set comprised 1,538 images, while our positive classification set included 372 distinct images of LeBron James. To ensure consistency across all inputs, we standardized the dataset by converting all images into square dimensions of 128 x 128 pixels. During this preprocessing step, we also centered the subject's face as much as possible within each frame to improve the model's ability to focus on distinguishing facial and contextual features. This normalization was crucial for enhancing training stability and enabling the model to generalize effectively across varying poses, lighting conditions, and backgrounds.

3 Models and Procedures

3.1 Baseline Model

For our baseline model, we used the most dominant class as our prediction. Based on our dataset of 1,538 images, consisting of 1,166 images labeled as "not LeBron" (including other NBA players, random people, and various objects) and 372 images labeled as "LeBron," the most frequent class was "not LeBron." By always predicting this negative class, we established a simple baseline accuracy. Since 1,166 out of 1,538 images are "not LeBron," our baseline accuracy is approximately 0.758. This gives us a reference point to evaluate the performance of more sophisticated models.

3.2 Standard CNN

Our Standard CNN expects input images of size 32×32 pixels. It consists of two main parts: a feature extractor and a classifier. The feature extractor is composed of four convolutional blocks, each containing a Conv2d layer (with increasing output channels: 32, 64, 128, and 256), followed by batch normalization, a ReLU activation, and a 2×2 max pooling layer. This sequence of layers reduces the spatial dimensions of the input from 32×32 down to 2×2. The classifier section begins with a flattening layer, followed by a fully connected (linear) layer that maps the 1024-dimensional input (256 channels

$\times$ 2 $\times$ 2) to 512 units, applies a ReLU activation, and then a dropout layer with a rate of 0.25. This is followed by another linear layer reducing the dimensionality to 256 units, again followed by a ReLU activation and dropout. The final linear layer outputs a single value (logit) for binary classification. The model is trained using the Adam optimizer with a learning rate of 1e-3 and the BCEWithLogitsLoss criterion for binary cross-entropy loss. This model architecture was heavily inspired by the CNNs used in lab section, this model is meant to be a base on which we can do further exploration as seen below.

3.3 Varying Image Complexity

Our first objective was to explore whether or not input image complexity affects model performance. We were curious about the results of this model since our original assumption was that increasing the image complexity would lead to more accurate results. This is based on the logic that, as humans, it is easier to identify people if more detail is provided. From our own personal testing, as we increased the resolution of the images, our ability to discern which images contained Lebron improved.

We introduced a new experimental setup where we perform a grid search over different input image sizes. Specifically we chose 16×16, 32×32, 64×64, and 128×128 pixels while keeping most of the aforementioned Standard CNN architecture and training configuration unchanged. This means for the most part, the convolutional layers, number of filters, activation functions, optimizer, and loss function remain the same as in our original model. The input image dimensions result in changing the layer sizes to accommodate the changed image dimensions.

Overall, this change allows us to systematically evaluate how the model adapts to varying levels of image detail and complexity in the task of classifying facial features.

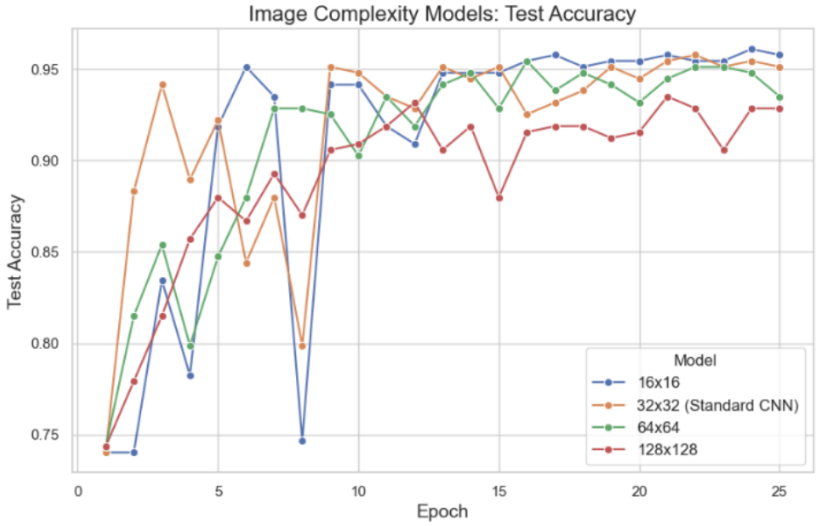

Figure 1: Test accuracy across different image sizes.

Looking at the graphing of our results, it shows that increasing image size doesn't always improve CNN performance. We can see that the general trend for all 4 variations are similar. Larger inputs may actually add unnecessary complexity without boosting accuracy, suggesting that for this task, low to moderate resolution is optimal for both accuracy and to reduce the overfitting of our model.

3.4 Hyperparameter Tuning

To create a more polished version of our Standard CNN architecture, we also implemented a simplistic hyperparameter tuning approach using grid search to identify optimal model configurations. While maintaining the same convolutional and fully connected layer structure as our Standard CNN, our grid search explores different combinations of three critical hyperparameters:

- **Learning Rate:** We test values ranging from `[1e-4, 5e-4, 1e-3, 5e-3]` to find the sweet spot between slow convergence and training instability.
- **Batch Size:** We explore batch sizes of `[16, 32, 64, 128]`, allowing us to evaluate the trade-off between training stability and memory requirements, as larger batches provide more stable gradients but also demand greater computational resources.

- **Dropout Rate:** We tested dropout values [0.1, 0.2, 0.3, 0.4] in both dropout layers simultaneously to examine how different levels of regularization affect the model's generalization capabilities.

This grid search approach evaluates 64 different model configurations (4x4x4) to identify the combination that yields the best performance. The total runtime of the grid search was 6 hours and 10 minutes. The process is computationally intensive but crucial for optimizing model performance, as these hyperparameters significantly impact training dynamics and final model accuracy. Given more time and computing power, we would ideally perform a more exhaustive tuning approach, encompassing more than 4 values per each hyperparameter.

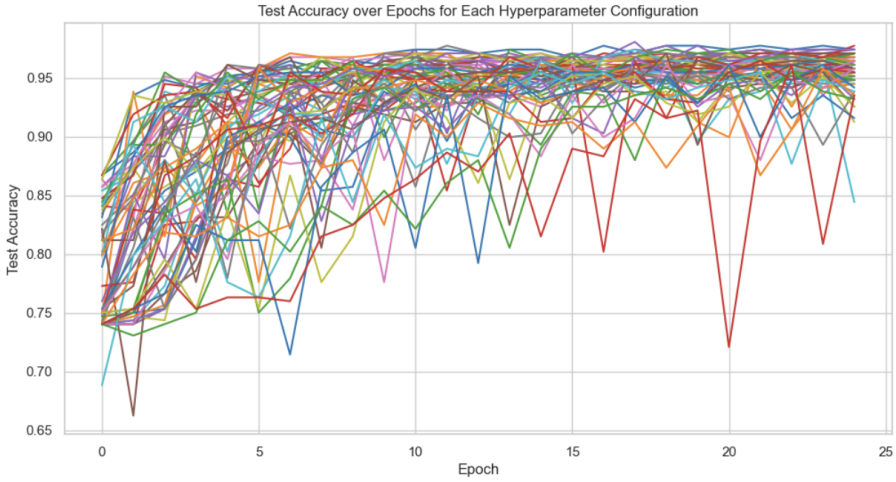

Figure 2: Hyperparameter tuning test accuracy overview across all configurations.

Learning Rate	Batch Size	Dropout Rate	Accuracy	TPR	FPR
1e-3	64	0.4	0.977	0.996	0.026
1e-4	16	0.1	0.974	0.987	0.033
1e-3	16	0.3	0.974	0.991	0.029
5e-4	64	0.1	0.974	0.991	0.029
1e-3	128	0.1	0.974	0.987	0.033

Table 1: Top 5 Performing Models After Hyperparameter Tuning

3.5 Loss Function Change

While maintaining the same Standard CNN architecture discussed initially, we implemented an alternative training configuration that replaced the binary cross-entropy with logits (BCEWithLogitsLoss) with standard cross-entropy loss. This alternative version maintains the identical convolutional and fully connected layer structure of our original model. Our main goal in implementing this change was to investigate changes in our training accuracy based on the two standard loss functoins used for binary classification.

Both models, whether trained with BCEWithLogitsLoss or CrossEntropyLoss, achieve similar training accuracy over the epochs. Accuracy rises quickly for both, surpassing 95 percent by around the 7th epoch, and both models reach nearly perfect accuracy (close to 100 percent) by epoch 20. While the Standard CNN with BCEWithLogitsLoss shows a slightly faster initial increase, the difference is minimal, and both models end up performing almost identically in terms of training accuracy.

Figure 3: Cross Entropy Loss model vs. Standard CNN.

4 Conclusion

4.1 Final Model Comparisons

Model	Accuracy	TPR	FPR	Precision	Number of Parameters
Baseline	0.758	0.93	0.91	0.51	0
Standard CNN (32x32)	0.95	0.95	0.067	0.957	1,045,761
Best Image Complexity Model (16x16)	0.96	0.96	0.028	0.956	667,137
Standard CNN with Cross-Entropy Loss	0.95	0.95	0.054	0.957	1,046,018
Hyperparameter-Tuned Standard CNN	0.977	0.996	0.026	0.962	1,045,261

Table 2: Model Performance Comparison

Our results clearly show that CNN-based models vastly outperform the baseline, confirming the strength of deep learning methods in binary image classification tasks. The baseline model, which had no trainable parameters and likely relied on simple heuristics or random guessing, achieved a modest accuracy of 75.8% with a high TPR of 0.93 but a low precision of 0.51. This implies that is produced a high number of false positives, making it unsuitable for real-world deployment where accurate identification is crucial.

Among the CNN models, the Standard CNN trained on 32×32 images set a strong baseline for deep learning approaches, with 95% accuracy and 0.957 precision. A variation trained with cross-entropy loss slightly improved the false positive rate to 0.054 while maintaining identical overall performance. The model trained on down-sampled 16×16 images emerged as a highly efficient option, achieving 96% accuracy with a significantly reduced parameter count (667,137) and an impressively low FPR of 0.028. This suggests that lower-resolution images may help the network focus on essential facial features and reduce overfitting while training.

The best performing model was the Standard CNN tuned to hyperparameters, which reached near-perfect accuracy (99.7%) and a TPR of 0.996, alongside the lowest FPR (0.026) and highest precision (0.962) across all models. This performance was achieved without increasing model complexity, as its parameter count (1,045,261) is comparable to the untuned CNN. This underscores the critical impact of tuning training parameters, such as learning rate, batch size, and optimizer, on model performance.

While all CNN variants significantly improved upon the baseline, the hyperparameter-tuned model offers the best results for high-stakes deployment where minimizing false positives is essential. However, the 16×16 image complexity model provides a strong trade-off between accuracy and computational efficiency, making it ideal for scenarios with limited hardware resources.

4.2 Model Bias and Implications

This project contributes to ongoing research in deep learning-based facial and object recognition by focusing on detecting LeBron James across varied visual environments. While the model may achieve high accuracy on training data, such metrics can mask underlying issues of overfitting and data bias. These biases limit the model's ability to generalize to

real-world scenarios, particularly when image conditions differ from those in the training set, leading to false positives or false negatives in practice.

A common source of bias is the over-representation of specific image types, such as high-resolution, front-facing game or press photos. This skews the model's learning and reduces robustness when handling side profiles, partial occlusions, or low-resolution images. In such cases, the model may confuse other players for LeBron or fail to recognize him altogether. Similarly, demographic and stylistic biases can arise if the dataset lacks diversity. For example, if LeBron is the only African American player or is mostly shown in certain team jerseys, the model may overfit to these attributes rather than learning meaningful facial or body features.

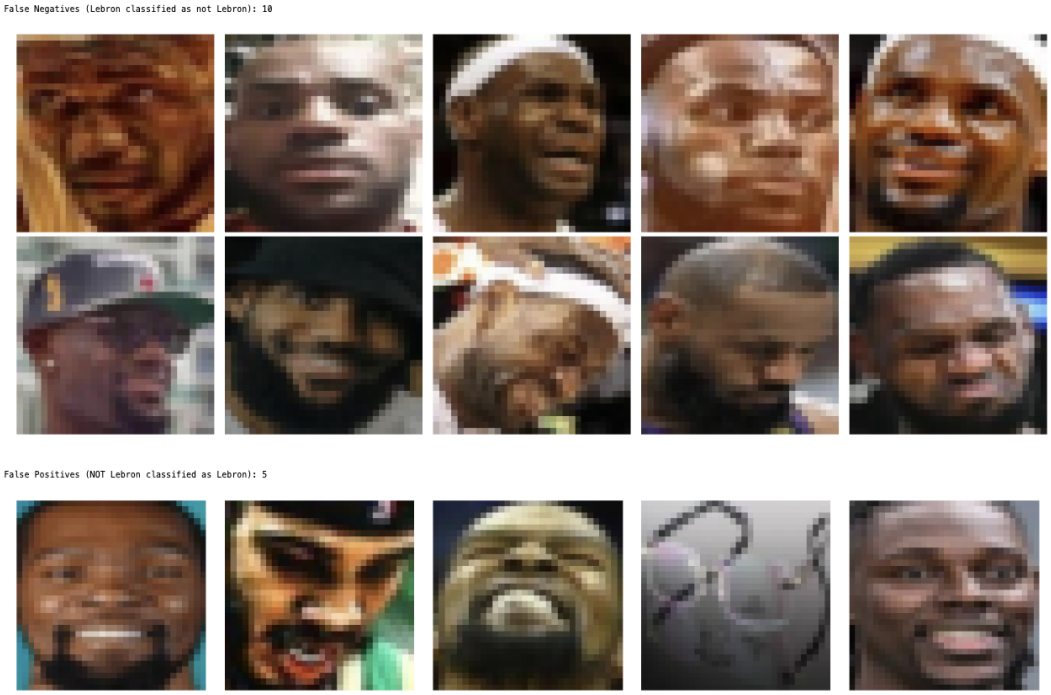

Figure 4: False negatives (top) and false positives (bottom) (Standard CNN)

The above visualizations of false negatives and false positives from the standard CNN provide clear qualitative insight into the model's limitations. Many of the false negatives (LeBron images classified as not LeBron) include atypical angles, occlusions, varied facial expressions, or accessories like hats and headbands. These variations deviate from the dominant training set distribution, which may have skewed toward more conventional or high-resolution portrayals. Conversely, the false positives (non-LeBron individuals misclassified as LeBron) often feature players with similar skin tone, facial hair, or expressions, indicating that the model may be relying on superficial visual cues rather than learning uniquely identifying features.

This reinforces the need for improved dataset diversity and augmentation strategies. Without sufficient exposure to a wide range of poses, lighting conditions, and contextual variations, even a high-performing CNN can exhibit brittle behavior when faced with real-world inputs. Future work could focus on enhancing the dataset through targeted collection of edge cases and leveraging techniques like hard example mining or adversarial training to reduce the likelihood of such misclassifications. Addressing these biases is critical not only for technical reliability but also for ethical model deployment. Inaccurate recognition has implications across applications like fan engagement, personalized sports highlights, media indexing, and live broadcasting. This work highlights that fair and representative training data is essential for building equitable and trustworthy AI systems.

5 Appendix

Please see HERE for our code zip file. All training data is already included, rerun notebook to reproduce results.

5.1 Standard CNN

```
#DATA PREPROCESSING
torch.manual_seed(42)

BATCH_SIZE = 64

transform = transforms.Compose([
    transforms.Resize(32), # <- IMAGE RESIZING TO 32x32
    transforms.ToTensor(),
    transforms.Normalize((0.5,)*3, (0.5,)*3)
])

data_root = "../../mega_batch"
full_ds   = datasets.ImageFolder(data_root, transform=transform)

# Calculate lengths for train and test splits
train_len = int(0.8 * len(full_ds))
test_len = len(full_ds) - train_len # Remaining samples for the test set

# Split the dataset
train_set, test_set = random_split(full_ds, [train_len, test_len])

# Create DataLoaders for each set
train_loader = DataLoader(train_set, batch_size=BATCH_SIZE, shuffle=True, num_workers=2, pin_memory=True)
test_loader = DataLoader(test_set, batch_size=BATCH_SIZE, shuffle=False, num_workers=2, pin_memory=True)

#CNN CLASS

class SimpleBinaryCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, 3, padding=1), nn.BatchNorm2d(32), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, 3, padding=1), nn.BatchNorm2d(64), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(64, 128, 3, padding=1), nn.BatchNorm2d(128), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(128, 256, 3, padding=1), nn.BatchNorm2d(256), nn.ReLU(),
            nn.MaxPool2d(2),
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(1024, 512), nn.ReLU(),
            nn.Dropout(0.25),
            nn.Linear(512, 256), nn.ReLU(),
            nn.Dropout(0.25),
            nn.Linear(256, 1)
        )

    def forward(self, x):
        return self.classifier(self.features(x))

# MODEL TRAINING

EPOCHS = 25
```

```
for epoch in range(EPOCHS):
    model.train()
    running_loss, correct = 0.0, 0

    for imgs, labels in train_loader:
        imgs, labels = imgs.to(device), labels.float().unsqueeze(1).to(device)

        optimizer.zero_grad()
        logits = model(imgs)
        loss = criterion(logits, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item() * imgs.size(0)
        preds = (torch.sigmoid(logits) > 0.5).float()
        correct += (preds == labels).sum().item()

    train_loss = running_loss / len(train_set)
    train_acc = correct / len(train_set)

    # --- test evaluation ---
    model.eval()
    test_correct = 0
    with torch.no_grad():
        for imgs, labels in test_loader:
            imgs, labels = imgs.to(device), labels.float().unsqueeze(1).to(device)
            logits = model(imgs)
            preds = (torch.sigmoid(logits) > 0.5).float()
            test_correct += (preds == labels).sum().item()
    test_acc = test_correct / len(test_set)

    # Store metrics
    train_losses.append(train_loss)
    train_accuracies.append(train_acc)
    test_accuracies.append(test_acc)
```

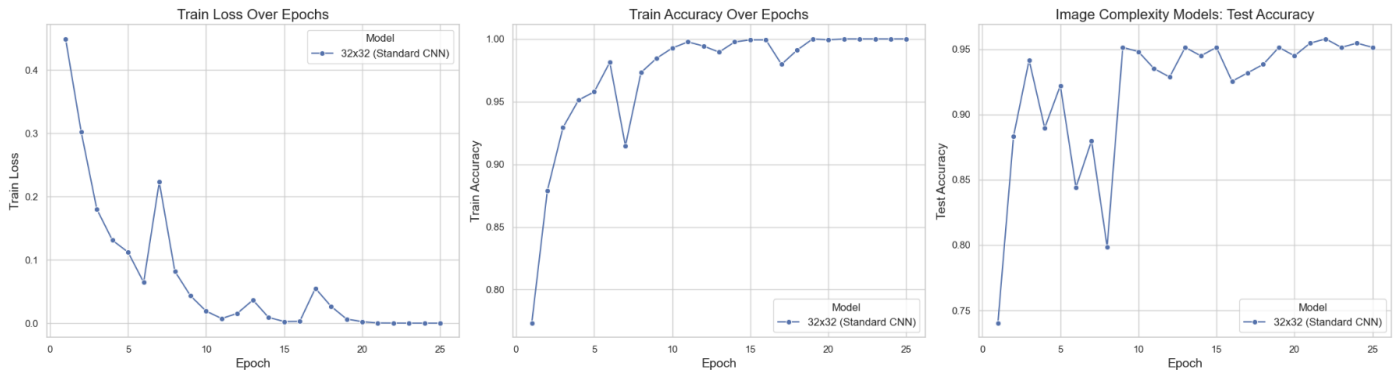

Figure 5: Standard CNN: Training Loss, Training Accuracy, and Test Accuracy

5.2 Varying Image Complexity Models

```
torch.manual_seed(42)

BATCH_SIZE = 64

transform = transforms.Compose([
    transforms.Resize((16, 16)),        # <- Images are resized to 16x16
    transforms.ToTensor(),
```

```
    transforms.Normalize((0.5,)*3, (0.5,)*3)
])

data_root = "../../mega_batch"
full_ds   = datasets.ImageFolder(data_root, transform=transform)

# Calculate lengths for train, validation, and test splits
train_len = int(0.8 * len(full_ds))
test_len = len(full_ds) - train_len

# Split the dataset
train_set, test_set = random_split(full_ds, [train_len, test_len])

# Create DataLoaders for each set
train_loader = DataLoader(train_set, batch_size=BATCH_SIZE, shuffle=True, num_workers=2, pin_memory=True)
test_loader = DataLoader(test_set, batch_size=BATCH_SIZE, shuffle=False, num_workers=2, pin_memory=True)

class SimpleBinaryCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, 3, padding=1), nn.BatchNorm2d(64), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(64, 128, 3, padding=1), nn.BatchNorm2d(128), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(128, 256, 3, padding=1), nn.BatchNorm2d(256), nn.ReLU(),
            nn.MaxPool2d(2),
        )

        # Manually calculated flattened size: 2 * 2 * 256 = 1024
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(2 * 2 * 256, 256), nn.ReLU(),  # <- Changed dimensions for compatibility with image size
            nn.Dropout(0.25),
            nn.Linear(256, 128), nn.ReLU(),
            nn.Dropout(0.25),
            nn.Linear(128, 1)
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x
```

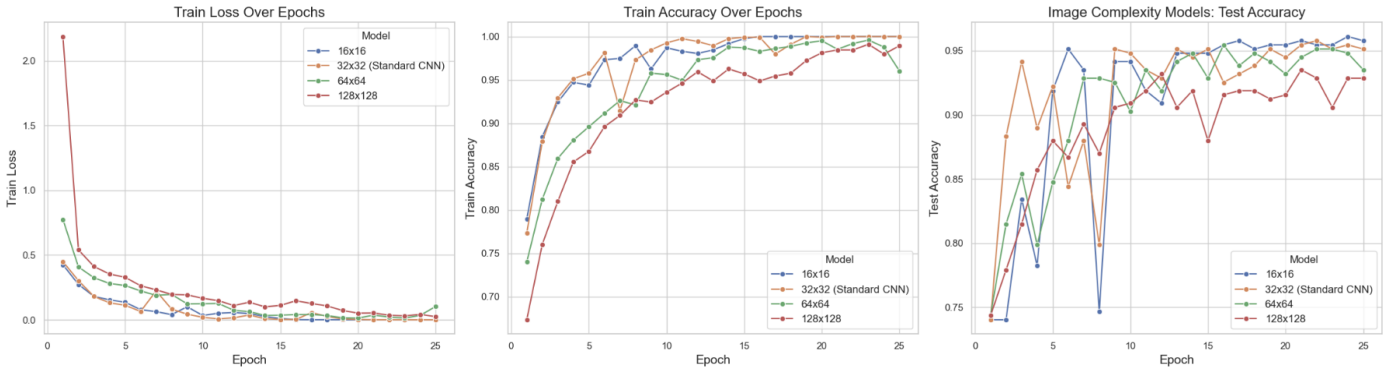

Figure 6: All 4 Image Size Models: Training Loss, Training Accuracy, and Test Accuracy

5.3 Hyperparameter Tuning

```
#SAME CNN ARCHITECTURE AS STANDARD CNN

#MODEL TRAINING FUNCTION

def train_model(batch_size, dropout_rate, learning_rate):
    # Update global variables with passed parameters
    global train_loader, test_loader, model, optimizer

    # Update DataLoader with new batch size
    train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True, num_workers=2, pin_memory=True)
    test_loader = DataLoader(test_set, batch_size=batch_size, shuffle=False, num_workers=2, pin_memory=True)

    # Update model and optimizer with new dropout rate
    model = SimpleBinaryCNN().to(device)
    model.classifier[3].p = dropout_rate # Update dropout rate
    optimizer = optim.Adam(model.parameters(), lr=learning_rate, weight_decay=1e-4)

    # Initialize lists to store metrics
    train_losses = []
    train_accuracies = []
    test_accuracies = []

    EPOCHS = 25

    for epoch in range(EPOCHS):
        model.train()
        running_loss, correct = 0.0, 0

        for imgs, labels in train_loader:
            imgs, labels = imgs.to(device), labels.float().unsqueeze(1).to(device)

            optimizer.zero_grad()
            logits = model(imgs)
            loss = criterion(logits, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item() * imgs.size(0)
            preds = (torch.sigmoid(logits) > 0.5).float()
            correct += (preds == labels).sum().item()

        train_loss = running_loss / len(train_set)
        train_acc = correct / len(train_set)

        # --- test evaluation ---
        model.eval()
        test_correct = 0
        true_labels, predicted_labels = [], []
        with torch.no_grad():
            for imgs, labels in test_loader:
                imgs, labels = imgs.to(device), labels.float().unsqueeze(1).to(device)
                logits = model(imgs)
                preds = (torch.sigmoid(logits) > 0.5).float()
                test_correct += (preds == labels).sum().item()
                true_labels.extend(labels.cpu().numpy())
                predicted_labels.extend(preds.cpu().numpy())
        test_acc = test_correct / len(test_set)

        # Store metrics
        train_losses.append(train_loss)
        train_accuracies.append(train_acc)
        test_accuracies.append(test_acc)
```

```
    # Calculate final metrics
    final_acc = test_accuracies[-1]
    true_positive = sum((true_labels[i] == 1 and predicted_labels[i] == 1) for i in range(len(true_labels)))
    false_positive = sum((true_labels[i] == 0 and predicted_labels[i] == 1) for i in range(len(true_labels)))
    false_negative = sum((true_labels[i] == 1 and predicted_labels[i] == 0) for i in range(len(true_labels)))
    true_negative = sum((true_labels[i] == 0 and predicted_labels[i] == 0) for i in range(len(true_labels)))

    tpr = true_positive / (true_positive + false_negative) if (true_positive + false_negative) > 0 else 0
    fpr = false_positive / (false_positive + true_negative) if (false_positive + true_negative) > 0 else 0

    return train_losses, train_accuracies, test_accuracies, final_acc, tpr, fpr

# GRID SEARCH

learning_rates = [0.0001, .0005, 0.001, 0.005]
batch_sizes = [16, 32, 64, 128]
dropout_rates = [0.1, 0.2, 0.3, 0.4]

# Initialize lists to store results
grid_search_results = []

for lr in learning_rates:
    for bs in batch_sizes:
        for dr in tqdm(dropout_rates, desc=f"Processing Dropout Rates for lr={lr}, bs={bs}"):
            model_name = f"model_lr_{lr}_bs_{bs}_dr_{dr}"
            # Update model and optimizer with current hyperparameters
            train_losses, train_accuracies, test_accuracies, final_acc, tpr, fpr = train_model(bs, dr, lr)

            # Store results for the current hyperparameter combination
            grid_search_results.append({
                'learning_rate': lr,
                'batch_size': bs,
                'dropout_rate': dr,
                'train_losses': train_losses,
                'train_accuracies': train_accuracies,
                'test_accuracies': test_accuracies,
                'final_acc': final_acc,
                'tpr': tpr,
                'fpr': fpr,
                'recall': recall
            })

df = pd.DataFrame(grid_search_results)

# Save as CSV
df.to_csv('grid_search_2.csv', index=False)
```

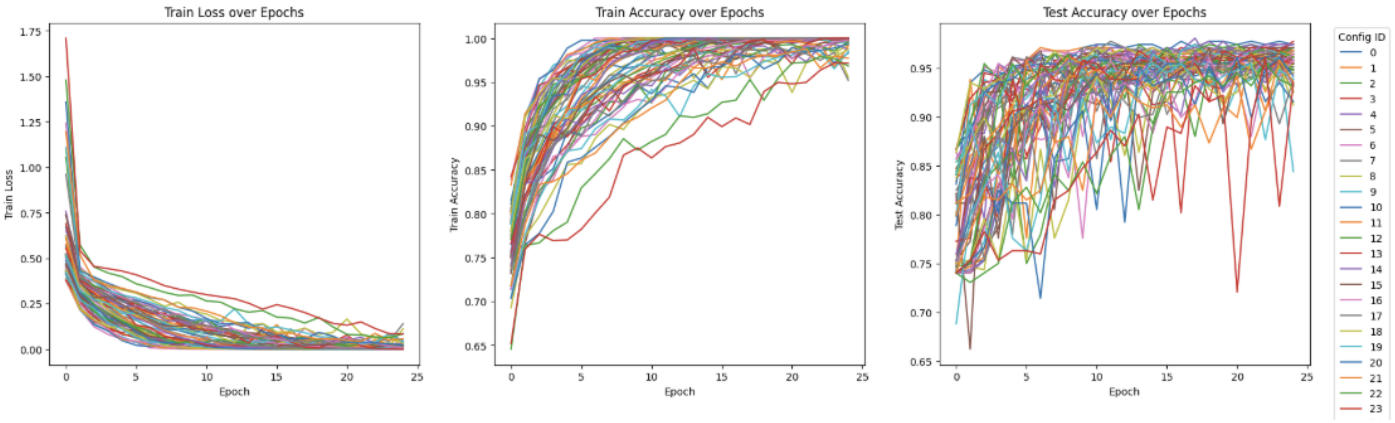

Figure 7: Hyperparameter Grid Search: Training Loss, Training Accuracy, and Test Accuracy

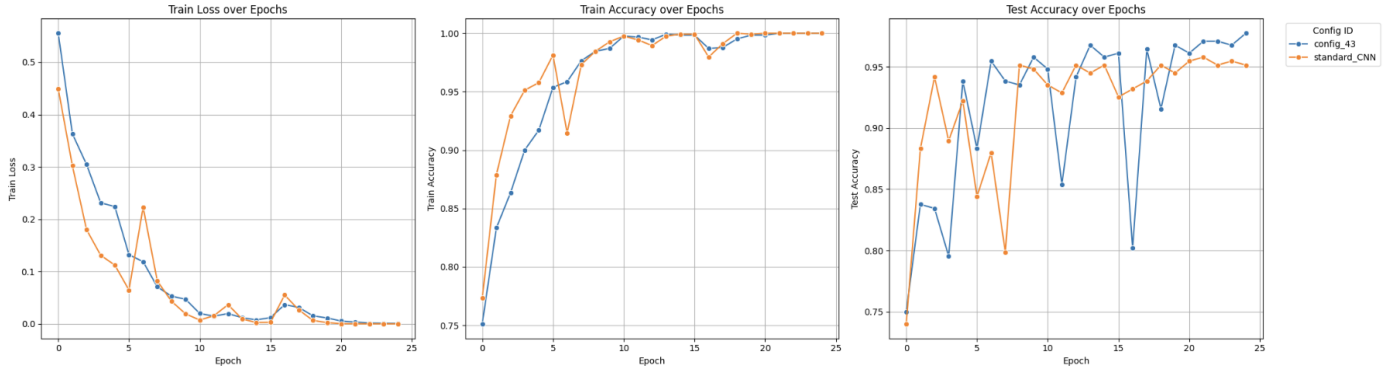

Figure 8: Tuned CNN vs Standard CNN: Training Loss, Training Accuracy, and Test Accuracy

5.4 Cross-Entropy Loss Model

```
learning_rate = 1e-3
criterion = nn.CrossEntropyLoss() #<- Loss function change
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

class SimpleBinaryCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, 3, padding=1), nn.BatchNorm2d(32), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, 3, padding=1), nn.BatchNorm2d(64), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(64, 128, 3, padding=1), nn.BatchNorm2d(128), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(128, 256, 3, padding=1), nn.BatchNorm2d(256), nn.ReLU(),
            nn.MaxPool2d(2),
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(1024, 512), nn.ReLU(),
            nn.Dropout(0.25),
            nn.Linear(512, 256), nn.ReLU(),
            nn.Dropout(0.25),
            nn.Linear(256, 2) # <- 2 output logits for CrossEntropyLoss
        )
```

```
    def forward(self, x):
        return self.classifier(self.features(x))
```

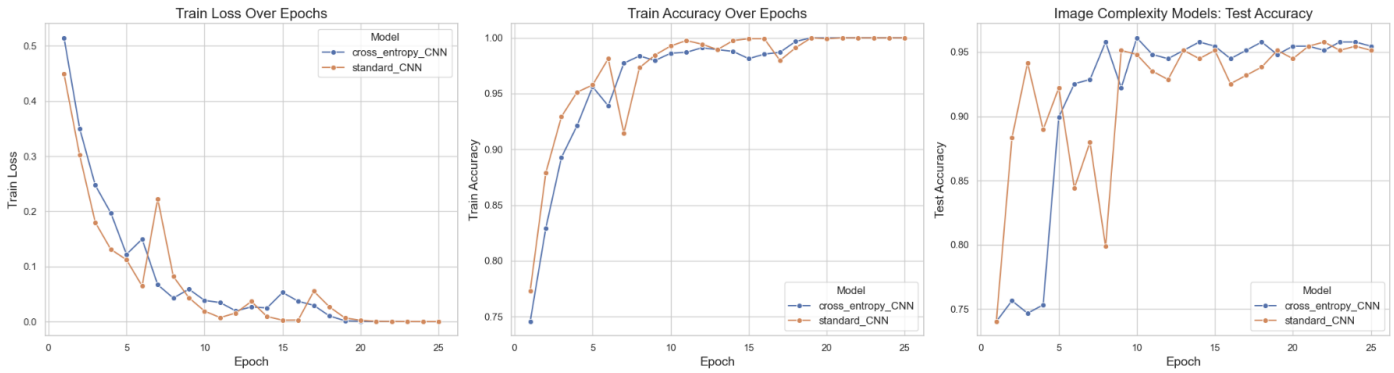

Figure 9: Cross-Entropy Loss Model: Training Loss, Training Accuracy, and Test Accuracy