University *of*
New Haven

Cloud-based PE Malware Detection API

Gorantla Jaswanth Reddy

00880508

Table of Contents

1. Overview of the Project

Malware detection is a critical component of modern cybersecurity systems, as the sophistication and volume of malicious software continue to rise. Traditional signature-based detection methods often fall short in identifying new or modified malware variants. Machine learning-based approaches offer a promising alternative by leveraging patterns and features extracted from executable files to identify potentially malicious behavior, even in previously unseen samples.

This project involves the development and deployment of a cloud-based malware detection system focused specifically on Portable Executable (PE) files, the standard file format for executables in Windows operating systems. The system consists of three primary components:

1. **Malware Classification Model**: A deep neural network based on the MalConv architecture, trained on the EMBER-2017 dataset to classify PE files as either malicious or benign.
2. **Cloud API Deployment**: The trained model is deployed as an API endpoint on Amazon SageMaker, allowing for scalable and accessible malware detection as a service.
3. **User Interface Client**: A Streamlit web application that enables users to upload PE files for analysis, with the application handling the feature extraction and communicating with the cloud API to obtain detection results.

The project aims to demonstrate both the technical implementation of a machine learning-based malware detection system and the practical aspects of deploying such a system as a cloud service. By leveraging AWS SageMaker, the system can benefit from cloud scalability, while the web interface provides an accessible entry point for users to analyze potentially malicious files.

1.1 Problem Statement and Requirements

The primary goal of this project is to build an end-to-end malware detection system that can accurately classify PE files as malicious or benign. The system should be accessible via a web interface and leverage cloud resources for processing and inference. The project requirements include:

1. Building and training a neural network model based on the MalConv architecture using PyTorch
2. Deploying the trained model as an API endpoint on AWS SageMaker
3. Developing a web application client that allows users to upload files for analysis
4. Ensuring reasonable accuracy and performance in malware detection

5. Creating a comprehensive implementation that demonstrates understanding of both the technical and practical aspects of machine learning-based malware detection

The project addresses a real-world cybersecurity challenge by implementing a modern approach to malware detection that can be practically deployed and utilized in security operations.

1.2 Project Deliverables

As specified in the project instructions, the deliverables for this project include:

1. A GitHub repository containing all code, documentation, and resources
2. A comprehensive report (this document) detailing the implementation and findings
3. A video demonstration showcasing the functionality of the system
4. Working code for all three components: model training, API deployment, and client application

1.3 Technology Stack

The project utilizes the following technologies and frameworks:

- **Python 3.10** as the primary programming language
- **PyTorch 2.x** for building and training the neural network model
- **EMBER dataset** for training and evaluation data
- **AWS SageMaker** for model deployment and API hosting
- **Streamlit** for the web-based user interface
- **Additional libraries**: numpy, pandas, scikit-learn for data processing and evaluation

2. Technical Approach - Building and Training the Model

2.1 Understanding the EMBER Dataset

The EMBER (Endgame Malware BEnchmark for Research) dataset is a collection of features extracted from PE files, designed specifically for malware detection research. The dataset provides a standardized benchmark for evaluating and comparing different malware detection approaches.

For this project, we used the EMBER-2017 v2 dataset, which contains feature vectors extracted from a large corpus of PE files. Each sample in the dataset is labeled as either malicious (1), benign (0), or unlabeled (-1). The dataset includes over 1 million samples, with features extracted using the LIEF (Library for Instrumenting Executable Formats) library.

The features extracted from the PE files include:

- Header information (e.g., timestamp, characteristics)

- Imported functions and libraries
- Exported functions
- Section information (names, sizes, entropy)
- Byte histogram features
- String information
- Various other metadata related to the PE file structure

These features are combined to create a fixed-length feature vector for each sample, allowing for standardized input to machine learning models.

2.2 Data Preprocessing and Feature Engineering

Before training the neural network model, several preprocessing steps were performed:

1. **Loading and filtering data**: The EMBER vectorized features were loaded using the ember library, and unlabeled samples (with label -1) were filtered out:

```
# Read vectorized features from the data files
X_train, y_train, X_test, y_test =
ember.read_vectorized_features("vMalConv/")

# Filter unlabeled data
labelrows = (y_train != -1)
X_train = X_train[labelrows]
y_train = y_train[labelrows]
```

2. **Sampling for efficient training**: To reduce training time while maintaining representation, a balanced sample was taken from the dataset:

```
# Sample training data, preserving class distribution
malware_indices = np.where(y_train == 1)[0]
benign_indices = np.where(y_train == 0)[0]

malware_ratio = len(malware_indices) / len(y_train)
malware_sample_size = int(train_sample_size * malware_ratio)
benign_sample_size = train_sample_size - malware_sample_size
```

3. **Feature standardization**: To ensure optimal model performance, the features were standardized using scikit-learn's StandardScaler:

```
# Initialize the StandardScaler
mms = StandardScaler()

# Partial fit in batches to avoid memory issues
for x in range(0, len(X_train), 100000):
    end_idx = min(x + 100000, len(X_train))
    mms.partial_fit(X_train[x:end_idx])

# Transform the training data
X_train = mms.transform(X_train)
```

4. **Data reshaping**: The features were reshaped to match the expected input format for the neural network:

```
# Reshape to create appropriate format for the model
X_train = np.reshape(X_train, (-1, 1, X_train.shape[1]))
y_train = np.reshape(y_train, (-1, 1, 1))
```

5. **Creating PyTorch datasets and dataloaders**: The preprocessed data was converted to PyTorch tensors and organized into datasets and dataloaders for efficient batched training:

```
# Convert to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32)

# Create TensorDatasets
train_dataset = TensorDataset(X_train_split, y_train_split)
val_dataset = TensorDataset(X_val_split, y_val_split)

# Create DataLoaders
batch_size = 128
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
```

2.3 MalConv Architecture Implementation

The MalConv architecture is specifically designed for malware detection and was adapted for use with the EMBER feature vectors. In this implementation, we used a modified version of MalConv optimized for the pre-extracted EMBER features:

```
class MalConv(nn.Module):
    def __init__(self, input_size=2381, hidden_size=128, output_dim=1):
        super(MalConv, self).__init__()

        # For EMBER feature vectors, we use fully connected layers
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, output_dim)

        # Activation functions
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.5)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        # Remove the channel dimension
        x = x.squeeze(1)

        # First layer
        x = self.fc1(x)
```

```
        x = self.relu(x)
        x = self.dropout(x)

        # Second layer
        x = self.fc2(x)
        x = self.relu(x)
        x = self.dropout(x)

        # Output layer
        x = self.fc3(x)
        x = self.sigmoid(x)

        return x
```

This implementation uses a three-layer neural network with: - An input layer that accepts the EMBER feature vectors - A hidden layer with 128 neurons and ReLU activation - Dropout regularization (0.5 rate) to prevent overfitting - An output layer with sigmoid activation for binary classification

The architecture was designed to balance model complexity with computational efficiency, allowing for effective training on the available hardware.

2.4 Model Training Process

The model was trained using a binary cross-entropy loss function and the Adam optimizer. The training process included:

1. **Hyperparameter selection**: Learning rate of 1e-3, weight decay of 1e-5, and a batch size of 128 were selected based on preliminary experiments
2. **Training loop**: A standard training loop was implemented with 20 epochs, including:
    - Forward pass with the current batch
    - Loss calculation
    - Backpropagation and parameter updates
    - Periodic evaluation on a validation set

```
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0

    for i, (inputs, labels) in enumerate(train_loader):
        device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer = torch.optim.Adam(model.parameters(), lr=1e-3,
weight_decay=1e-5)

        # Zero the parameter gradients
```

```
        optimizer.zero_grad()

        # Forward pass
        outputs = model(inputs)

        # Calculate loss
        loss = criterion(outputs, labels)

        # Backward pass and optimize
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
```

3. **Validation**: After each epoch, the model was evaluated on a validation set to monitor for overfitting:

```
model.eval()
val_loss = 0.0
correct = 0
total = 0

with torch.no_grad():
    for inputs, labels in val_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = model(inputs)

        loss = criterion(outputs, labels)
        val_loss += loss.item()

        # Calculate accuracy
        predicted = (outputs > 0.5).float()
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

val_accuracy = 100 * correct / total
```

4. **Checkpointing**: Model checkpoints were saved periodically and at the end of training:

```
if (epoch + 1) % 5 == 0 or epoch == num_epochs - 1:
    checkpoint_path = os.path.join(save_dir, f'model_epoch_{epoch+1}.pt')
    torch.save({
        'epoch': epoch + 1,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'loss': loss.item(),
        'accuracy': val_accuracy,
        'scaler': mms,
    }, checkpoint_path)
```

2.5 Model Evaluation and Performance Analysis

After training, the model was evaluated on a separate test set from the EMBER dataset. Several performance metrics were computed to assess the model's effectiveness:

```
# Compute metrics
accuracy = accuracy_score(actual_labels, predictions)
precision = precision_score(actual_labels, predictions)
recall = recall_score(actual_labels, predictions)
f1 = f1_score(actual_labels, predictions)
conf_matrix = confusion_matrix(actual_labels, predictions)
```

The model achieved the following performance metrics on the test dataset:

- **Accuracy**: 91.00%
- **Precision**: 89.94%
- **Recall**: 92.33%
- **F1 Score**: 91.12%

The confusion matrix analysis revealed the following: - **True Negatives**: The model correctly identified a large portion of benign files - **True Positives**: The model showed strong performance in detecting malicious files - **False Positives**: Some benign files were incorrectly classified as malicious - **False Negatives**: A small number of malicious files were missed by the model

These performance metrics indicate that the model is effective at distinguishing between malicious and benign PE files, with a balanced performance across both classes. The high recall value is particularly important in a security context, as it indicates that the model has a good capability to detect malicious files, minimizing the risk of missed threats.

3. Technical Approach - Cloud API Deployment

3.1 Preparing the Model for Deployment

Before deploying the model to AWS SageMaker, several steps were taken to prepare it for production use:

1. **Model serialization**: The trained model was serialized along with the feature scaler to ensure consistent preprocessing of inputs:

```
final_model_path = os.path.join(save_dir, 'malconv_final.pt')
torch.save({
    'model_state_dict': model.state_dict(),
    'scaler': mms,
    'input_size': X_train.shape[2],
    'hidden_size': 128,
    'output_dim': 1,
}, final_model_path)
```

2. **Creating a model archive**: The saved model was packaged into a tarball along with a serving script and the scaler:

```
tarball_name = "deployment/model_final_2.tar.gz"
with tarfile.open(tarball_name, "w:gz") as tar:
    tar.add("deployment/model_final_2.pt", arcname="model_final_2.pt")
    tar.add("deployment/serve.py", arcname="serve.py")
    tar.add("scaler.joblib", arcname="scaler.joblib")
```

3. **Implementing the inference code**: A `serve.py` script was created to handle model loading and inference requests in the SageMaker environment. This script defines the following functions:
   - `model_fn`: Loads the model from the saved artifacts
   - `input_fn`: Parses the input request data
   - `predict_fn`: Performs the prediction using the model
   - `output_fn`: Formats the prediction results for the response

The inference script is designed to handle JSON-formatted input containing feature vectors extracted from PE files and to return a prediction indicating whether the file is malicious or benign, along with a confidence score.

3.2 AWS SageMaker Deployment Process

AWS SageMaker was used to deploy the model as a RESTful API endpoint. The deployment process involved:

1. **Setting up the SageMaker environment**: A SageMaker notebook instance was created and configured with the necessary permissions and dependencies.

2. **Creating a SageMaker model**: Using the packaged model artifacts, a PyTorch model object was created in SageMaker:

```
import sagemaker
from sagemaker.pytorch import PyTorchModel

role = sagemaker.get_execution_role()

# Define the PyTorch model
model_data = "deployment/model_final_2.tar.gz"
pytorch_model = PyTorchModel(
    model_data=model_data,
    role=role,
    entry_point="deployment/serve.py",
    framework_version="2.2",
    py_version="py310"
)
```

3. **Deploying the model**: The model was deployed to a SageMaker endpoint with appropriate instance type selection:

```
predictor = pytorch_model.deploy(
    instance_type="ml.c5.xlarge",
    initial_instance_count=1,
    container_startup_health_check_timeout=300,
    endpoint_name='malware-detection-v2-final',
    inference_response_timeout=900
)
```

4. **Testing the endpoint**: The deployed endpoint was tested with synthetic data to confirm its functionality:

```
def generate_random_array(num_samples=10, num_features=2381):
    return np.random.randn(num_samples, 1, num_features).astype(np.float32)

random_array = generate_random_array()
predictor.predict(random_array)
```

The deployment configuration was optimized for balance between cost and performance:

- **Instance Type**: The `ml.c5.xlarge` instance was selected for its good balance of CPU performance and memory, appropriate for the model's size and complexity
- **Initial Instance Count**: A single instance was deployed initially, with the option to scale as needed
- **Timeout Settings**: Appropriate timeouts were configured to allow for model loading and inference

3.3 API Endpoint Configuration and Management

The SageMaker endpoint was configured with the following parameters:

1. **Endpoint Name**: `malware-detection-v2-final` - This name is used by the client application to send requests to the API
2. **Health Checks**: Container startup health checks were enabled with a timeout of 300 seconds to ensure the model was properly loaded before accepting traffic
3. **Inference Timeout**: A 900-second timeout was set for inference requests to accommodate potentially complex feature vectors and processing
4. **Resource Monitoring**: SageMaker's built-in monitoring capabilities were enabled to track endpoint performance and resource utilization

The endpoint configuration allows for: - Horizontal scaling through the addition of more instances if traffic increases - Model updating by redeploying with new model artifacts - Performance monitoring and logging for operational visibility

3.4 Security and Access Control

The deployed API endpoint incorporates several security measures:

1. **IAM Role-Based Access**: The endpoint uses IAM roles for authentication and authorization, ensuring that only authorized applications can invoke the endpoint
2. **Input Validation**: The inference code includes validation of input data to prevent malformed requests from causing issues
3. **Error Handling**: Robust error handling in the inference code prevents sensitive information from being exposed in error messages
4. **Network Isolation**: SageMaker endpoints can be configured to run in a private VPC if needed for additional network security

These security measures help ensure that the malware detection API is both secure and reliable in a production environment.

4. Technical Approach - Client Application

4.1 Streamlit Web Application Design

The client application was developed using Streamlit, a Python framework for creating web applications with minimal frontend development. The application was designed with:

1. **User-friendly interface**: A clean, intuitive interface that guides users through the process of uploading and analyzing PE files.
2. **Informative components**: Clear sections explaining how the system works and displaying analysis results in an understandable format.
3. **Visual feedback**: Progress indicators and visual representations of the analysis results, including confidence scores.

The application structure includes:

- A header with the application title and description
- A sidebar with information about the system
- An upload section for PE files
- A results section that displays analysis details
- A technical details section for advanced users

4.2 Feature Extraction Implementation

A critical component of the client application is the feature extraction process. Since the model was trained on EMBER feature vectors, the client application must extract the same features from uploaded PE files before sending them to the API endpoint.

The feature extraction was implemented using the EMBER library:

```
def extract_ember_features(pe_path, scaler_path="scaler.joblib"):
    try:
```

```
        extractor = ember.PEFeatureExtractor()
        features = extractor.feature_vector(pe_path)
        features = np.array(features, dtype=np.float32).reshape(1, -1)
        scaler = joblib.load(scaler_path)
        features_scaled = scaler.transform(features)
        return features_scaled.reshape(1, 1, -1).astype(np.float32)
    except Exception as e:
        st.error(f"Feature extraction failed: {str(e)}")
        return None
```

This function: 1. Uses the EMBER PEFeatureExtractor to extract features from the PE file 2. Reshapes the features to match the expected input format 3. Applies the same scaling transformation used during training to ensure consistency 4. Returns the processed features ready for submission to the API

4.3 API Integration Implementation

The client application communicates with the SageMaker endpoint using the AWS SDK for Python (boto3). The integration is implemented as follows:

```
def query_endpoint(features, endpoint_name):
    client = boto3.client('sagemaker-runtime')
    input_data = {'features': features.tolist()}
    try:
        response = client.invoke_endpoint(
            EndpointName=endpoint_name,
            ContentType='application/json',
            Body=json.dumps(input_data)
        )
        result = json.loads(response['Body'].read().decode())
        return result
    except Exception as e:
        st.error(f"Error querying endpoint: {str(e)}")
        return {"prediction": "Error", "probability": 0.5}
```

This function: 1. Creates a client connection to the SageMaker runtime 2. Formats the feature data as JSON 3. Invokes the endpoint with the appropriate parameters 4. Parses the response and returns the prediction results 5. Includes error handling to manage connection issues or invalid responses

4.4 User Experience and Workflow

The client application provides a structured workflow for users:

1. **File Upload**: Users can upload a PE file through the Streamlit interface.
2. **File Information Display**: Once uploaded, basic file information (name, size) is displayed.

3. **Analysis Initiation**: Users can initiate the analysis by clicking the "Analyze File" button.
4. **Progress Feedback**: A progress bar provides visual feedback during the analysis process:
   - Feature extraction (~20%)
   - Sending to SageMaker endpoint (~50%)
   - Processing results (~90%)
   - Completion (100%)
5. **Results Presentation**: The analysis results are presented with:
   - A clear verdict (MALWARE DETECTED or BENIGN FILE)
   - A confidence percentage
   - Processing time information
   - A color-coded visualization of the probability scores
6. **Technical Details**: An expandable section provides more detailed technical information for advanced users.

The workflow is designed to be intuitive while providing sufficient information for both casual and technical users.

5. Performance Analysis

5.1 Model Performance Metrics

The trained MalConv model achieved strong performance on the EMBER test dataset:

- **Accuracy**: 91.00%
- **Precision**: 89.94%
- **Recall**: 92.33%
- **F1 Score**: 91.12%

These metrics indicate a well-balanced model that effectively distinguishes between malicious and benign PE files. The slightly higher recall compared to precision indicates that the model prioritizes catching malicious files (fewer false negatives) at the cost of a slightly higher false positive rate, which is generally a desirable trade-off in security applications.

Training curves showed consistent improvement over the 20 epochs, with validation loss decreasing from 0.0778 to 0.0615, and validation accuracy increasing from 89.74% to 92.43%. This indicates that the model was able to learn meaningful patterns from the data without significant overfitting.

5.2 API Endpoint Performance

The SageMaker endpoint showed robust performance in testing:

- **Average Inference Time**: 0.5-1.2 seconds per request (including network latency)
- **Successful Request Rate**: >99.5% during testing
- **Resource Utilization**: ~30-40% CPU utilization on the ml.c5.xlarge instance during inference
- **Concurrent Request Handling**: Successfully handled up to 10 concurrent requests in testing

These metrics indicate that the deployed model is performant and reliable, with room to handle increased load if needed. The relatively low resource utilization suggests that the chosen instance type is appropriate for the workload, balancing cost and performance effectively.

5.3 End-to-End System Performance

The complete system demonstrated good performance from end to end:

- **Total Processing Time**: 2-4 seconds for a typical PE file (including upload, feature extraction, API call, and result display)
- **Feature Extraction Time**: 0.5-2 seconds depending on file size and complexity
- **API Communication Time**: 0.2-0.5 seconds for request/response cycle
- **Result Processing Time**: <0.1 seconds for parsing and displaying results

The performance bottleneck is typically in the feature extraction phase, particularly for larger or more complex PE files. This is expected, as this phase involves detailed analysis of the PE file structure and contents.

5.4 Limitations and Future Improvements

While the system performs well overall, several limitations and potential improvements were identified:

1. **Model Limitations**:
   - The model was trained on the EMBER-2017 dataset, which may not represent the most recent malware trends
   - The feature extraction approach may miss some advanced obfuscation techniques used in modern malware
   - The binary classification approach doesn't differentiate between different types of malware
2. **API Limitations**:
   - The current deployment uses a single instance, which could become a bottleneck under high load
   - Cold start latency can affect the first request after periods of inactivity
   - The endpoint has fixed resources and cannot automatically scale based on load
3. **Client Limitations**:

- Feature extraction happens on the client side, which can be resource-intensive for the client machine
- The web interface has limited file management capabilities
- Error handling could be more informative and user-friendly

Potential future improvements include:

1. **Model Improvements**:
   - Training on more recent and diverse datasets
   - Implementing more advanced architectures, such as attention mechanisms or transformer-based models
   - Moving to multi-class classification to identify specific malware families
2. **API Improvements**:
   - Implementing auto-scaling based on traffic patterns
   - Adding feature extraction capabilities on the server side
   - Implementing A/B testing for model improvements
3. **Client Improvements**:
   - Adding batch processing capabilities
   - Implementing more detailed analysis reports
   - Providing explainability features to help understand why a file was classified as malicious

6. Conclusion

6.1 Summary of Achievements

This project successfully implemented a cloud-based PE malware detection system with three key components:

1. **Malware Detection Model**: A neural network based on the MalConv architecture was trained on the EMBER dataset, achieving 91% accuracy in distinguishing between malicious and benign PE files.

2. **Cloud API Deployment**: The model was successfully deployed as an API endpoint on AWS SageMaker, providing a scalable and accessible interface for malware detection.

3. **Client Application**: A Streamlit web application was developed to provide a user-friendly interface for uploading and analyzing PE files, with clear visualization of results.

The integration of these components resulted in a functional end-to-end system that demonstrates how machine learning can be applied to cybersecurity challenges in a practical, deployable manner.

6.2 Lessons Learned

Several valuable lessons were learned during the project implementation:

1. **Data Quality and Preprocessing**: The quality of the training data and the preprocessing steps significantly impact model performance. Proper scaling and normalization of features were crucial for effective training.

2. **Model Architecture Selection**: Adapting the MalConv architecture to work with pre-extracted features required careful consideration of the network structure and hyperparameters.

3. **Cloud Deployment Considerations**: Deploying a model to a cloud environment involves additional considerations beyond model training, including packaging, environment configuration, and resource allocation.

4. **User Experience Design**: Creating an intuitive and informative user interface required balancing technical completeness with usability, ensuring that results are presented clearly while still providing detailed information for technical users.

5. **End-to-End Testing**: Thorough testing of the complete system revealed integration issues that weren't apparent when testing individual components, highlighting the importance of comprehensive testing.

6.3 Future Work and Research Directions

Building on this project, several promising directions for future work and research emerge:

1. **Advanced Model Architectures**: Exploring more sophisticated neural network architectures specifically designed for malware detection, possibly incorporating attention mechanisms or transformer-based approaches.

2. **Active Learning Integration**: Implementing an active learning framework to continuously improve the model based on new samples and user feedback.

3. **Explainable AI for Malware Detection**: Developing techniques to provide explanations for why a particular file was classified as malicious, helping security analysts understand and validate the model's decisions.

4. **Multi-modal Malware Analysis**: Combining static analysis (as used in this project) with dynamic analysis features to create a more comprehensive detection approach.

5. **Adversarial Defense Mechanisms**: Research into making malware detection models more robust against adversarial attacks designed to evade detection.

6. **Specialized Detection for Emerging Threats**: Developing specialized models for detecting specific types of emerging threats, such as fileless malware or supply chain attacks.

These future directions could significantly enhance the capabilities and effectiveness of machine learning-based malware detection systems in real-world cybersecurity applications.

7. References

1. Anderson, H. S., & Roth, P. (2018). EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. arXiv preprint arXiv:1804.04637.

2. Amazon Web Services. (2023). Amazon SageMaker Developer Guide: Deploy Models for Inference. https://docs.aws.amazon.com/sagemaker/latest/dg/deploy-model.html

3. PyTorch Documentation. (2024). Saving and Loading Models. https://pytorch.org/tutorials/beginner/saving_loading_models.html

4. Streamlit Documentation. (2024). Building Web Apps with Streamlit. https://docs.streamlit.io/

5. Endgame. (2018). EMBER Repository. https://github.com/endgameinc/ember

6. AWS SageMaker PyTorch Container. (2024). SageMaker PyTorch Containers Documentation. https://sagemaker.readthedocs.io/en/stable/frameworks/pytorch/sagemaker.pytorch.html