# System Threat Forecaster: Complete Technical Workflow

## Machine Learning & Deep Learning Approaches

Technical Documentation

December 19, 2025

# Agenda

# Project Overview

## Objective

Predict potential malware infections in computer systems using machine learning and deep learning approaches

**Dataset:**

- 100,000 training samples
- 76 features (48 numeric, 28 categorical)
- Binary classification (malware vs clean)
- Balanced classes: 50.52% malware, 49.48% clean

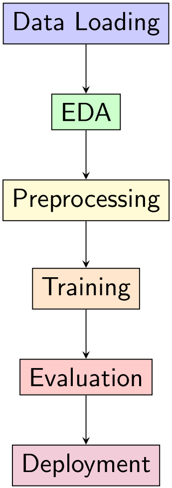
**Approaches:**

- Traditional ML: 7 algorithms
- Deep Learning: 6 neural architectures
- Modular pipeline design
- Production-ready implementation

# Key Features

1. **Modular Architecture**: Independent components
2. **Multiple Models**: Compare 7 ML + 6 DL algorithms
3. **Automated Pipeline**: End-to-end workflow
4. **Flexible Configuration**: Central CONFIG system
5. **Comprehensive Logging**: Track experiments
6. **Web App Integration**: Deployment-ready models

# ML Workflow: Architecture Overview



# ML: Environment Setup

```python
import numpy as np, pandas as pd
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, accuracy_score
from sklearn.ensemble import RandomForestClassifier
from lightgbm import LGBMClassifier
```

## Key Libraries

**Scikit-learn** (ML algorithms), **LightGBM** (best performer), **Pandas/NumPy** (data), **Matplotlib** (visualization)

# ML: Configuration System

```
CONFIG = {
    'data_path': {'train': './kaggle/input/train.csv',
                  'test': './kaggle/input/test.csv'},
    'run_eda': True,
    'models_to_train': {
        'decision_tree': True,
        'random_forest': True,
        'lightgbm': True,  # Best performer
        'ada_boost': True
    },
    'random_state': 42
}
```

# ML: Configuration Benefits

**Centralized Control**

Single point to enable/disable pipeline components

**Data Pipeline Control:**

- File paths
- Missing value handling
- Feature engineering
- Feature selection
- Dimensionality reduction

**Model Selection:**

- 7 algorithms available
- Easy enable/disable
- Hyperparameter tuning
- Cross-validation folds
- Train-test split ratio

# ML: Utility Functions

```python
def load_data(path):
    return pd.read_csv(path)

def save_model(model, model_name):
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    filename = f"models/{model_name}_{timestamp}.joblib"
    joblib.dump(model, filename)

def log_result(model_name, metrics, hyperparams=None):
    log_data = {'model_name': model_name,
                'accuracy': metrics['accuracy'],
                'timestamp': datetime.now()}
    df.to_csv('results/model_comparison.csv', mode='a')
```

# ML: Utility Benefits

1. **Data Management**: Consistent loading, error handling
2. **Model Persistence**: Timestamped files for versioning
3. **Results Logging**: Performance tracking and model selection

# ML: EDA Overview

**Purpose**

Understand dataset characteristics to guide preprocessing and model selection

**Dataset Analysis:**

- Shape: 100,000 $\times$ 76
- Data types identification
- Missing value patterns
- Statistical summaries

**Feature Analysis:**

- Distribution plots
- Correlation heatmap
- Target variable balance
- Top correlated features

# ML: Key EDA Findings

- **Features**: 48 numeric, 28 categorical
- **Target**: Balanced (50.52% vs 49.48%)
- **Missing Values**: Several columns affected
- **Top Correlations**: AntivirusConfigID (0.118), TotalPhysicalRAMMB (0.066)
- **Insight**: Moderate correlations → ensemble models needed

# ML: Preprocessing Pipeline

1. **Feature Separation**: Split X, y and identify column types
2. **Missing Values**: Mean (numeric), mode (categorical)
3. **Encoding**: LabelEncoder for categorical features
4. **Scaling**: StandardScaler: $z = \frac{x-\mu}{\sigma}$

# ML: Preprocessing Code

```python
def preprocess_data(data, is_training=True):
    X = data.drop('target', axis=1)
    y = data['target']

    numeric_cols = X.select_dtypes(include=['int64']).columns
    categorical_cols = X.select_dtypes(include=['object']).columns

    X[numeric_cols] = SimpleImputer(strategy='mean').fit_transform(X[numeric_cols])
    X[categorical_cols] = SimpleImputer(strategy='most_frequent').fit_transform(X[categorical_cols])

    X = LabelEncoder().fit_transform(X)
    X_scaled = StandardScaler().fit_transform(X)

    return X_scaled, y
```

# ML: Optional Feature Processing

### Feature Engineering (Disabled)

Creates interaction features (e.g., AntivirusConfigID $\times$ TotalPhysicalRAMMB)

### Feature Selection (Disabled)

SelectKBest with ANOVA F-test keeps top 30 features

### Dimensionality Reduction (Disabled)

PCA with 95% variance $\rightarrow$ $\sim$40 components

# ML: Seven Algorithms Implemented

| Model | Accuracy | Type |
|---|---|---|
| LightGBM | **63.0%** | Gradient Boosting |
| Random Forest | 62.4% | Ensemble (Bagging) |
| AdaBoost | 62.0% | Ensemble (Boosting) |
| Decision Tree | ∼60% | Single Tree |
| Logistic Regression | ∼60% | Linear Model |
| Naive Bayes | Lower | Probabilistic |
| SGD | Lower | Online Learning |

## Best Model

**LightGBM** achieves 63.0% accuracy due to:

- Leaf-wise tree growth (vs level-wise)
- Handles large datasets efficiently
- Built-in categorical feature support

# ML: Training Process

```python
def train_model(model_name, X_train, y_train, X_val, y_val):
    params = get_default_model_params(model_name)

    model = get_model(model_name, params)
    model.fit(X_train, y_train)

    val_pred = model.predict(X_val)
    metrics = {
        'val_accuracy': accuracy_score(y_val, val_pred),
        'precision': precision_score(y_val, val_pred),
        'f1_score': f1_score(y_val, val_pred)
    }

    save_model(model, model_name)
    return model, metrics
```

# ML: Evaluation Metrics

**Accuracy**:

$$\frac{\text{Correct Predictions}}{\text{Total Predictions}}$$

**Precision**:

$$\frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

**Recall**:

$$\frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

**F1-Score**:

$$2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

## Confusion Matrix

Visualizes True Positives, False Positives, True Negatives, False Negatives

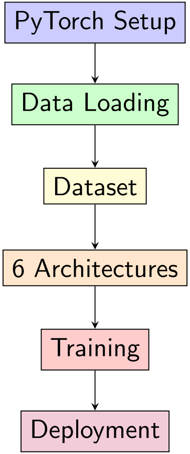
# ML: Complete Pipeline Flow

1. Load `train.csv` and `test.csv`
2. Run EDA and preprocess (handle missing, encode, scale)
3. Split train (80%) / validation (20%)
4. Optional: Feature engineering, selection, PCA
5. Train all selected models (7 algorithms)
6. Compare performance and select best
7. Generate predictions and save models
8. Export metrics to `results/model_performance.json`

# ML: Output Files

| File | Purpose |
|---|---|
| saved_models/ml_models.pkl | All trained models |
| saved_models/preprocessors.pkl | Scalers, encoders |
| results/model_comparison.csv | Performance log |
| results/model_performance.json | Structured metrics |
| submission.csv | Kaggle submission |
| models/lightgbm_*.joblib | Versioned models |

# DL Workflow: Architecture Overview



# DL: Environment Setup

```python
import torch, torch.nn as nn
from torch.utils.data import Dataset, DataLoader

torch.manual_seed(42)  # Reproducibility

# Device detection (GPU acceleration)
if torch.cuda.is_available():
    device = torch.device('cuda')  # NVIDIA
elif torch.backends.mps.is_available():
    device = torch.device('mps')   # Apple Silicon
else:
    device = torch.device('cpu')   # CPU
```

## GPU Acceleration

CUDA (10-100×), MPS (5-15×), CPU (1×)

# DL: Configuration

```
CONFIG = {
    'batch_size': 512,
    'epochs': 100,
    'learning_rate': 0.001,
    'hidden_dims': [256, 128, 64, 32],
    'dropout_rate': 0.3,
    'use_batch_norm': True,
    'optimizer': 'adamw',
    'use_mixed_precision': torch.cuda.is_available()
}
```

# DL: Configuration Comparison

| Setting | ML | DL |
|---|---|---|
| Training | One-shot `.fit()` | Iterative epochs |
| Batch Size | N/A | 512 |
| Learning Rate | N/A | 0.001 |
| Regularization | N/A | Dropout (0.3) |
| Data Format | DataFrame | Tensors |
| Feature Eng. | Manual | Automatic |
| Computation | CPU | GPU-accelerated |
| Training Time | Minutes | Hours |

# DL: Custom Dataset Class

```python
class TabularDataset(Dataset):
    def __init__(self, X, y=None):
        self.X = torch.FloatTensor(X)
        self.y = torch.LongTensor(y) if y else None

    def __len__(self): return len(self.X)
    def __getitem__(self, idx):
        return (self.X[idx], self.y[idx]) if self.y else self.X[idx]

train_loader = DataLoader(train_dataset, batch_size=512,
                          shuffle=True, num_workers=4)
```

# DL: DataLoader Benefits

1. **Batching**: Groups 512 samples, memory efficient
2. **Shuffling**: Prevents order patterns, improves generalization
3. **Parallel Loading**: 4 workers eliminate bottleneck
4. **Class Imbalance**: Weighted loss [1.98, 2.02]

# DL: Six Architectures

| Model | Accuracy | Time | Params |
|---|---|---|---|
| Simple MLP | 60-62% | 5 min | 50K |
| Deep MLP | 62-64% | 10 min | 100K |
| Residual Net | 63-65% | 15 min | 200K |
| Attention Net | 64-66% | 20 min | 300K |
| Wide & Deep | 63-65% | 12 min | 150K |
| FT-Transformer | **65-68%** | 30 min | 500K |

### Best Architecture

**FT-Transformer**: 4-5% improvement over traditional ML

# Architecture 1: Simple MLP

**Structure**:

- Input (76 features)
- Linear($76 \rightarrow 256$) + ReLU + Dropout
- Linear($256 \rightarrow 128$) + ReLU + Dropout
- Linear($128 \rightarrow 64$) + ReLU + Dropout
- Output($64 \rightarrow 2$)

**Components**:

- **Linear**: $y = Wx + b$
- **ReLU**: $\max(0, x)$
- **Dropout**: Zeros 30% neurons

**Use Case**:

- Baseline model
- Fast training
- 60-62% accuracy

# Architecture 2: Deep MLP with Batch Norm

```python
class DeepMLP(nn.Module):
    def __init__(self, input_dim, hidden_dims=[256, 128, 64]):
        super().__init__()
        layers = []
        for hidden_dim in hidden_dims:
            layers.append(nn.Linear(prev_dim, hidden_dim))
            layers.append(nn.BatchNorm1d(hidden_dim))
            layers.append(nn.ReLU())
            layers.append(nn.Dropout(0.3))
        self.network = nn.Sequential(*layers)
```

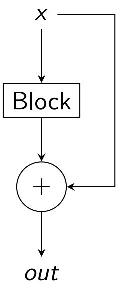
## Batch Normalization

$$x_{norm} = \frac{x - \mu_{batch}}{\sqrt{\sigma^2_{batch} + \epsilon}}, \quad output = \gamma \cdot x_{norm} + \beta$$

Faster training, stability (62-64% accuracy)

# Architecture 3: Residual Network

**Skip Connections**:




$$out = Block(x) + x$$

**Benefits**:

- Solves vanishing gradients
- Deeper networks (100+ layers)
- 63-65% accuracy

# Architecture 4: Attention Network

**Self-Attention**: $\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) \cdot V$

**Mechanism**:

- Each feature attends to others
- Learns interactions automatically

**Benefits**:

- Complex relationships
- Interpretable weights
- 64-66% accuracy

**Innovation**

Transforms NLP architecture for tabular data

# Architecture 5: Wide & Deep

**Simplified Architecture:**

Input (76 features) splits into:

- Wide path: Direct Linear(76 $\rightarrow$ 2)
- Deep path: MLP 256 $\rightarrow$ 128 $\rightarrow$ 64 $\rightarrow$ 2

Both outputs are added together for final prediction.

**Components**:

- **Wide**: Memorization (frequent patterns)
- **Deep**: Generalization (novel combinations)

**Use Case**:

- Google Play recommendations
- 63-65% accuracy

# Architecture 6: FT-Transformer (Best)

**Feature Tokenizer Transformer** - SOTA for tabular data

1. **Tokenization**: Each feature $\rightarrow$ 192-dim vector
2. **CLS Token**: Prepend classification token (like BERT)
3. **Positional Embeddings**: Learnable position encoding
4. **Transformer**: 8-head self-attention, 3 blocks
5. **Classification**: CLS token $\rightarrow$ Linear(192 $\rightarrow$ 2)

**Performance**

**65-68% accuracy** - 4-5% improvement over ML

# DL: Training Loop

1. **Forward**: $\hat{y} = f_\theta(x)$ (network prediction)
2. **Loss**: $L = -\log(P(\text{correct class}))$ (CrossEntropy)
3. **Backward**: $\frac{\partial L}{\partial \theta}$ (compute gradients)
4. **Update**: $\theta_{t+1} = \theta_t - \alpha \cdot \nabla_\theta L$ (AdamW)

**Iterative Learning**

Repeat for 100 epochs

# DL: Training Code

```python
def train_epoch(model, train_loader, criterion, optimizer):
    model.train()

    for data, target in train_loader:
        output = model(data)
        loss = criterion(output, target)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    return avg_loss, accuracy

for epoch in range(100):
    train_loss = train_epoch(...)
    val_loss = validate(...)
    if early_stopping(val_loss): break
```

# DL: Advanced Training Techniques

1. **Early Stopping**: Stop if no improvement for 15 epochs
2. **LR Scheduling**: Reduce when stuck ($0.001 \rightarrow 0.0005$)
3. **Mixed Precision**: FP16 for 2-3$\times$ speedup (NVIDIA)
4. **Checkpointing**: Save best model by validation accuracy

# DL: Complete Pipeline Flow

1. Load and preprocess $\rightarrow$ NumPy arrays
2. Create DataLoaders (batch size=512)
3. Split train (80%) / validation (20%)
4. Train 6 architectures (MLP, ResNet, Attention, FT-Transformer)
5. Evaluate and select best model (FT-Transformer ~67%)
6. Generate predictions and save models
7. Export to `submission_dl.csv`

# ML vs DL: Comprehensive Comparison

| **Aspect** | **Machine Learning** | **Deep Learning** |
|---|---|---|
| Best Model | LightGBM (63.0%) | FT-Transformer (65-68%) |
| Training | One-shot `.fit()` | Iterative epochs |
| Data Format | DataFrame | Tensors |
| Feature Eng. | Manual | Automatic |
| Architecture | Fixed (trees) | Flexible (networks) |
| Computation | CPU-optimized | GPU-accelerated |
| Training Time | 5-15 minutes | 30-60 minutes |
| Interpretability | High | Low (black box) |
| Hyperparams | Few ($\sim$10) | Many ($\sim$50) |
| Memory | Low | High (GPU) |

# Performance Summary

| Category | Model | Accuracy |
|---|---|---|
| Traditional ML | LightGBM | **63.0%** |
| | Random Forest | 62.4% |
| | AdaBoost | 62.0% |
| Deep Learning | FT-Transformer | **65-68%** |
| | Attention Net | 64-66% |
| | Residual Net | 63-65% |

**Key Finding**

Deep Learning achieves **4-5% improvement** over traditional ML

# When to Use ML vs DL

**Use Traditional ML When:**

- Small dataset (¡10K samples)
- Need interpretability
- Limited compute resources
- Quick prototyping
- Tabular data with clear features
- Need fast inference

**Use Deep Learning When:**

- Large dataset (¿50K samples)
- Complex patterns
- GPU available
- Maximum performance needed
- Many feature interactions
- Time for experimentation

**Our Project**

With 100K samples and GPU access, DL provides the best performance

# Deployment Pipeline

1. **Model Saving**: ML (7 models), DL (6 architectures), preprocessors
2. **Metadata**: Best model, accuracy, hyperparameters
3. **Web App**: Load preprocessors and best model (FT-Transformer)
4. **Inference**: User input $\rightarrow$ Preprocess $\rightarrow$ Model $\rightarrow$ Prediction

# Web App Integration Code

```python
import joblib, torch

preprocessors = joblib.load('saved_models/preprocessors.pkl')
checkpoint = torch.load('saved_models/dl_models.pth')
model = checkpoint['best_model']
model.eval()

def predict(input_features):
    X = preprocessors['scaler'].transform([input_features])

    with torch.no_grad():
        output = model(torch.FloatTensor(X))
        pred = output.argmax(dim=1).item()

    return "Malware" if pred == 1 else "Clean"
```

# Key Achievements

1. **Implementation**: 7 ML + 6 DL algorithms, production-ready
2. **Performance**: ML (63.0%), DL (65-68%), 4-5% improvement
3. **Technical**: GPU acceleration, mixed precision, Transformers
4. **Deployment**: Model versioning, web app, comprehensive logging

# Technical Highlights

## Machine Learning

- Modular configuration system
- Multiple algorithm comparison
- Comprehensive preprocessing pipeline
- Feature engineering capabilities

## Deep Learning

- 6 state-of-the-art architectures
- Transformer-based approach for tabular data
- Advanced training techniques (early stopping, LR scheduling)
- GPU acceleration support

# Future Improvements

1. **Ensemble**: Combine ML + DL predictions, stacking
2. **Optimization**: Optuna, Ray Tune, NAS
3. **Features**: Cross-validation, advanced imputation
4. **Deployment**: ONNX export, quantization, A/B testing

# Thank You

## Questions?

**Repository Structure:**

- `system-threat-forecaster-ml.py` - ML Pipeline
- `system-threat-forecaster-dl.py` - DL Pipeline
- `saved_models/` - Trained models
- `results/` - Performance metrics
- `models/` - Model checkpoints

**Best Performance:** FT-Transformer at 65-68% accuracy

# Appendix: Hyperparameters - LightGBM

| Parameter | Value |
|---|---|
| n_estimators | 200 |
| learning_rate | 0.1 |
| max_depth | 5 |
| subsample | 0.6 |
| colsample_bytree | 0.8 |
| min_child_samples | 20 |
| reg_alpha | 0.1 |
| reg_lambda | 0.1 |

# Appendix: Hyperparameters - FT-Transformer

| Parameter | Value |
|---|---|
| d_token | 192 |
| n_blocks | 3 |
| attention_heads | 8 |
| attention_dropout | 0.2 |
| ffn_dropout | 0.1 |
| residual_dropout | 0.0 |
| learning_rate | 0.001 |
| weight_decay | 1e-5 |
| batch_size | 512 |
| epochs | 100 |

# Appendix: Dataset Features

**Numeric Features (48)**:

- TotalPhysicalRAMMB
- AvailablePhysicalRAMMB
- ProcessorCount
- SystemVolumeTotalCapacity
- SystemDriveFreeSpace
- ...

**Categorical Features (28)**:

- AntivirusConfigID
- FirewallConfigID
- OSVersion
- CountryIdentifier
- LocaleEnglishName
- ...

**Target Variable**: Binary (0=Clean, 1=Malware)