A Beginner's Guide to

Hardware Trojan Detection Using Deep Learning

A Step-by-Step Tutorial for Undergraduate Students

Prerequisites: Basic Python programming, familiarity with arrays/matrices **No prior knowledge required of:** Deep learning, Verilog, Hardware security **Estimated reading time:** 2-3 hours

Date: January 26, 2026

What You'll Learn

Part 1: Understanding the Problem (30 min)

• What are integrated circuits?

• What is a hardware trojan?

• Why is detection important?

Part 2: Deep Learning Basics (45 min)

• What is machine learning?

• Neural networks explained simply

• How computers 'learn' patterns

Part 3: Our Three Solutions (60 min)

• Graphs and Graph Neural Networks

• Transformers (like ChatGPT!)

• CNNs and LSTMs combined

Part 4: Hands-On Code Walkthrough (45 min)

• Setting up your environment

• Running the code step-by-step

• Understanding the results

Part 1: Understanding the Problem

1.1 What is an Integrated Circuit (IC)?

An **Integrated Circuit** (IC), also called a "chip" or "microchip", is a small electronic device made of semiconductor material (usually silicon). It contains thousands to billions of tiny components like transistors, resistors, and capacitors. **You use ICs every day:** • Your smartphone has several ICs (processor, memory, sensors) • Your laptop's CPU is a very complex IC • Credit cards have small ICs for security • Cars have dozens of ICs controlling various systems Think of an IC as a tiny city of electronic components, all working together to process information or perform specific tasks.

1.2 How are ICs Designed?

IC design happens in several stages: **1. Specification:** Define what the chip should do **2. RTL Design:** Write code describing the chip's behavior using a Hardware Description Language (HDL) like **Verilog**. This is similar to programming, but describes hardware instead of software. **3. Synthesis:** Convert the code into actual circuit components (gates) **4. Fabrication:** Manufacture the physical chip in a factory (foundry) **The Problem:** Companies often outsource fabrication to other countries because building a chip factory costs billions of dollars. This creates opportunities for malicious modifications!

Here's a simple Verilog example - a 2-input AND gate:

```
module and_gate ( input wire a, // First input input wire b, // Second input output wire y
// Output: a AND b ); assign y = a & b; // & means AND operation endmodule // When a=1 and
b=1, then y=1 // Otherwise y=0
```

Figure: A simple AND gate in Verilog

1.3 What is a Hardware Trojan?

A **Hardware Trojan** is a malicious modification to an integrated circuit. Just like the famous Trojan Horse from Greek mythology that hid soldiers inside a gift, a hardware trojan hides malicious functionality inside a seemingly normal chip. **Two Main Parts: 1. Trigger:** The condition that activates the trojan. Like a time bomb waiting for a specific moment. Examples: • Counter reaching a specific value (activate after 1 million operations) • Specific input sequence (like a secret password) • Particular date/time **2. Payload:** The malicious action. What happens when triggered: • Leak secret data (passwords, encryption keys) • Cause the chip to malfunction • Create a backdoor for attackers

Real-World Impact: A hardware trojan in a military system could leak classified information. In a medical device, it could cause life-threatening malfunctions. In financial systems, it could steal money. This is why detection is critical!

1.4 Example: A Simple Hardware Trojan

Let's look at a simple example. Here's a normal adder (adds two numbers):

```
// CLEAN VERSION - Normal 4-bit adder module adder ( input [3:0] a, input [3:0] b, output [4:0] sum ); assign sum = a + b; // Simply add a and b endmodule
```

Now here's the SAME adder with a trojan hidden inside:

```
// TROJANED VERSION - Adder with hidden malicious code module adder ( input [3:0] a, input [3:0] b, output [4:0] sum ); // TROJAN: Counter that increments every operation reg [15:0] counter = 0; always @(a or b) counter <= counter + 1; // TROJAN: After 50000 operations, output wrong result! wire trojan_active = (counter > 50000); // Normal operation OR trojan operation assign sum = trojan_active ? 5'b11111 : (a + b); endmodule // The chip works perfectly for 50000 uses, then fails!
```

Why is this dangerous? • The trojan is invisible during normal testing (works fine initially) • After 50,000 uses, it starts giving wrong answers • If this adder is used in a bank's calculator, imagine the errors! • The extra code (counter) is hidden among legitimate code

Part 2: Deep Learning Basics

2.1 What is Machine Learning?

Traditional Programming: You give the computer explicit rules. Example: "If email contains 'lottery' AND 'winner', mark as spam" **Machine Learning:** You give the computer examples, and it learns the rules itself! Example: Show the computer 10,000 spam emails and 10,000 normal emails. It figures out what makes an email spam. **Analogy:** Teaching a child vs. giving instructions to a robot • Robot: "Walk 3 steps, turn left 90 degrees, walk 5 steps..." • Child: Show them how to walk, let them practice, they learn! Machine Learning is like teaching a child - you show examples, not explicit rules.

2.2 What is a Neural Network?

A **Neural Network** is inspired by how our brain works. Your brain has billions of neurons (brain cells) connected to each other. When you see a cat, certain neurons activate, and you recognize "that's a cat!" **Artificial Neural Networks work similarly:** 1. **Input Layer:** Receives data (like pixels of an image) 2. **Hidden Layers:** Process the data, finding patterns Each "neuron" does a simple calculation and passes result to next layer 3. **Output Layer:** Gives the answer (like "cat" or "dog") **Simple Math Inside Each Neuron:** output = activation(weight1 × input1 + weight2 × input2 + ... + bias) The "weights" are numbers the network learns during training!

Inputs	Weights	Calculation	Output
x1 = 0.5	w1 = 0.4	sum = 0.5×0.4 + 0.8×0.6	
x2 = 0.8	w2 = 0.6	= 0.2 + 0.48 = 0.68	
bias = 0.1		total = 0.68 + 0.1 = 0.78	ReLU(0.78) = 0.78

Table: Example calculation in one neuron

2.3 How Does a Neural Network Learn?

Training Process (simplified): Step 1: Forward Pass Feed an example through the network, get a prediction. Example: Show an image, network says "dog" (but it's actually a cat!) **Step 2: Calculate Error (Loss)** Compare prediction with correct answer. Error = how wrong was the prediction? **Step 3: Backward Pass (Backpropagation)** Figure out which weights caused the error. "These weights made us say dog instead of cat" **Step 4: Update Weights** Adjust weights slightly to reduce error. Do this for thousands of examples! **Analogy:** Like adjusting a recipe after each attempt: "Too salty? Use less salt next time!"

Key Terms: • **Epoch:** One complete pass through all training data • **Batch:** Small group of examples processed together • **Learning Rate:** How big each weight adjustment is • **Loss Function:** Measures how wrong predictions are • **Optimizer:** Algorithm that adjusts weights (e.g., Adam, SGD)

2.4 Different Types of Neural Networks

Different problems need different network architectures: **1. Feedforward Networks (Basic)** Data flows in one direction: input → hidden layers → output Good for: Simple classification, regression **2. Convolutional Neural Networks (CNNs)** Special layers that scan across data looking for patterns Like sliding a magnifying glass across an image Good for: Images, any grid-like data **3. Recurrent Neural Networks (RNNs) / LSTMs** Have "memory" - can remember previous inputs Good for: Sequences (text, time series, code!) **4. Transformers** Use "attention" - can look at all parts of input simultaneously Powers ChatGPT, Google Translate, and many modern AI systems Good for: Text, code, anything with long-range dependencies **5. Graph Neural Networks (GNNs)** Work on graph-structured data (nodes connected by edges) Good for: Social networks, molecules, circuit diagrams!

Part 3: Our Three Solutions Explained

3.1 Solution 1: Graph Neural Network (GNN)

The Idea: Verilog code describes how signals connect to each other - this is naturally a GRAPH! A graph has nodes (things) and edges (connections). **Example:** In the code "assign y = a & b;", we have: • Node: a (input signal) • Node: b (input signal) • Node: y (output signal) • Edge: a → y (a affects y) • Edge: b → y (b affects y) **Why This Works for Trojans:** Trojans add extra signals and connections that look suspicious! A GNN learns what "normal" connection patterns look like, then flags abnormal patterns. **Real-World Analogy:** Think of a social network. Normal friend connections form certain patterns. A spy network would have unusual patterns - people connected in strange ways. GNNs can detect these abnormal patterns!

How GNN Works (Simplified): 1. Each signal (node) starts with a description (features): - Is it an input? Output? Wire? Register? - How many bits wide is it? - How often is it used? 2. Each node "talks" to its neighbors: "Hey neighbors, what are your features?" Combines neighbor info with its own. 3. Repeat step 2 several times: Now each node knows about neighbors-of-neighbors! 4. Combine all node information: Create one summary for the whole circuit. 5. Classify: "Does this summary look like a trojan or clean design?"

3.2 Solution 2: Transformer

The Idea: Treat Verilog code like a sentence in English. Just like ChatGPT reads sentences to understand meaning, we read Verilog to detect trojans. **Key Concept: Attention** When reading "The cat sat on the mat", to understand "sat", you pay attention to "cat" (who sat) and "mat" (where). In Verilog: "assign y = a & b;" To understand "y", pay attention to "a" and "b" (what determines y). **Why This Works for Trojans:** Trojan code has unusual attention patterns! Normal code: signals attend to related signals Trojan code: suspicious connections to unrelated signals **Real-World Analogy:** Reading a book vs. a book with hidden messages. Normal book: sentences make sense, words relate naturally Hidden message: certain words don't fit, unusual patterns Transformers detect these unusual patterns!

How Transformer Works (Simplified): 1. **Tokenization:** Break code into words (tokens) "assign y = a & b;" → ["assign", "y", "=", "a", "&", "b", ";"] 2. **Embedding:** Convert each word to numbers Each token becomes a vector (list of numbers) 3. **Self-Attention:** Each token looks at ALL other tokens Asks: "Which other tokens are relevant to me?" Creates attention weights (importance scores) 4. **Multiple Layers:** Repeat attention several times Each layer understands more complex patterns 5. **Classification:** Use final representation to decide "Trojan" or "Clean"

3.3 Solution 3: Hybrid CNN-LSTM

The Idea: Combine the strengths of two different approaches: • CNN: Great at finding local patterns (like specific code snippets) • LSTM: Great at understanding sequences (order matters!) **CNN Part - Finding Local Patterns:** Slides a "window" across the code looking for suspicious patterns. Like searching a document for certain phrases. Different window sizes catch different patterns: - Small window: operators, simple expressions - Large window: entire statements, blocks **LSTM Part - Understanding Sequence:** Reads code left-to-right AND right-to-left (bidirectional). Remembers what it saw earlier. "This signal was defined earlier, now it's being used strangely..." **Why This Works for Trojans:** Trojans often have: - Specific patterns (caught by CNN) - Unusual sequential relationships (caught by LSTM)

How Hybrid Model Works (Simplified): 1. **Feature Extraction:** - Convert code to numbers (like Transformer) - Also extract statistics: line count, signal count, etc. 2. **Multi-Scale CNN:** - Use 5 different window sizes: 3, 5, 7, 11, 15 - Each finds patterns of different scales - Combine all findings 3. **BiLSTM:** - Read CNN output left-to-right (forward) - Read CNN output right-to-left (backward) - Combine both directions 4. **Attention:** - Learn which parts of code are most important - Weight important parts more 5. **Fusion & Classification:** - Combine sequence features with statistics - Make final prediction

3.4 Comparing the Three Solutions

Aspect	GNN	Transformer	Hybrid CNN-LSTM
Best at	Structure	Long patterns	Balanced
Speed	Fast	Slow	Medium
Memory	Low	High	Medium
Complexity	Medium	High	Medium
Analogy	Social network	Reading book	Scanning + Reading

Part 4: Hands-On Code Walkthrough

4.1 Setting Up Your Environment

Step 1: Install Python Download Python 3.8 or newer from python.org **Step 2: Create a Virtual Environment (Recommended)**

```
# Open terminal/command prompt # Create virtual environment python -m venv trojan_env # Activate it: # On Windows: trojan_env\Scripts\activate # On Mac/Linux: source trojan_env/bin/activate # Install required packages pip install torch numpy scikit-learn matplotlib pip install torch-geometric # For GNN solution pip install reportlab # For PDF generation
```

4.2 Understanding the Project Structure

After downloading, you'll see this folder structure:

```
HW_Trojan_Detection_Solutions/ ■ ■■■ solution1_gnn/ # Graph Neural Network solution ■ ■■■ gnn_trojan_detector.py ■ ■■■ solution2_transformer/ # Transformer solution ■ ■■■ transformer_trojan_detector.py ■ ■■■ solution3_cnn_lstm/ # Hybrid solution ■ ■■■ cnn_lstm_trojan_detector.py ■ ■■■ requirements.txt # List of packages needed ■■■ run_all_solutions.py # Run everything at once ■■■ generate_documentation.py # Create these PDFs!
```

4.3 Running Your First Detection

Let's run the simplest solution (Hybrid CNN-LSTM):

```
# Navigate to the project folder cd HW_Trojan_Detection_Solutions # Run the hybrid solution python solution3_cnn_lstm/cnn_lstm_trojan_detector.py # You'll see output like: # Loading dataset... # Loaded 236 files # Clean: 208, Trojaned: 28 # Vocabulary size: 3500 # # Starting training... # Epoch 1/100 # Train Loss: 0.6823, Train Acc: 0.5500 # Val Loss: 0.6102, Val Acc: 0.6800 # Epoch 2/100 # Train Loss: 0.5234, Train Acc: 0.7200 # ...
```

Understanding the Output: • **Loss:** How wrong the model is (lower = better) Started at 0.68, decreased to ~0.2 = model is learning! • **Accuracy:** Percentage of correct predictions Started at 55%, improved to ~90% = great performance! • **Epoch:** One complete pass through training data More epochs = more learning (but don't overdo it!) • **Train vs Val:** Train = learning data, Val = testing data (model hasn't seen) If Val accuracy is much lower than Train, model is "memorizing" not "learning"

4.4 Key Code Sections Explained

Let's look at the most important parts of the code: **1. Loading Verilog Files:**

```
def _load_dataset(self): # Walk through all folders for root, dirs, files in
os.walk(self.data_dir): for file in files: if file.endswith('.v'): # Verilog files
filepath = os.path.join(root, file) # Determine label from folder name label = 0 # Clean by
default if 'TjIn' in root: # TjIn folder = trojaned label = 1
self.file_paths.append(filepath) self.labels.append(label)
```

2. The Neural Network Model:

```
class HybridCNNLSTMTrojanDetector(nn.Module): def __init__(self, vocab_size=5000,
hidden_dim=256): super().__init__() # Convert word IDs to vectors self.embedding =
nn.Embedding(vocab_size, 128) # CNN for local patterns self.cnn =
MultiScaleCNN(input_dim=128, hidden_dim=256) # LSTM for sequential understanding self.lstm
= BiLSTMEncoder(input_dim=256, hidden_dim=256) # Final classification layer
self.classifier = nn.Sequential( nn.Linear(256, 128), nn.ReLU(), nn.Linear(128, 2) # 2
classes: clean, trojaned ) def forward(self, x): x = self.embedding(x) # Words → Vectors x
= self.cnn(x) # Find local patterns x = self.lstm(x) # Understand sequence x =
x.mean(dim=1) # Average across sequence return self.classifier(x) # Predict class
```

3. Training Loop:

```
def train_epoch(self, train_loader): self.model.train() # Enable training mode for batch
in train_loader: # Get data inputs = batch['sequence'].to(self.device) labels =
batch['label'].to(self.device) # Forward pass self.optimizer.zero_grad() # Clear old
gradients predictions = self.model(inputs) # Calculate loss loss =
self.criterion(predictions, labels) # Backward pass loss.backward() # Calculate gradients
self.optimizer.step() # Update weights
```

4.5 Interpreting Results

After training completes, you'll see a classification report:

```
Classification Report: precision recall f1-score support Clean 0.95 0.98 0.96 42 Trojaned
0.86 0.75 0.80 8 accuracy 0.94 50 macro avg 0.90 0.86 0.88 50 weighted avg 0.93 0.94 0.93
50
```

What These Numbers Mean: • **Precision (0.86 for Trojaned):** When the model says "trojaned", it's right 86% of the time. (14% false alarms) • **Recall (0.75 for Trojaned):** Of all actual trojans, the model catches 75%. (25% missed - this is more concerning!) • **F1-Score:** Combined measure of precision and recall. Higher = better balance between the two. • **Accuracy (0.94):** Overall, 94% of predictions are correct. But be careful - this can be misleading with unbalanced data! **For Security:** High recall is crucial - we'd rather have false alarms than miss real trojans! A missed trojan could be catastrophic.

Glossary of Terms

Attention: Mechanism that allows neural networks to focus on relevant parts of input

Backpropagation: Algorithm for calculating gradients to update network weights

Batch: Group of training examples processed together

CNN: Convolutional Neural Network - good at detecting local patterns

Epoch: One complete pass through all training data

GNN: Graph Neural Network - works on graph-structured data

Gradient: Direction and magnitude of weight adjustments

Hardware Trojan: Malicious modification to an integrated circuit

LSTM: Long Short-Term Memory - type of RNN with better memory

Loss: Measure of how wrong the model's predictions are

Neuron: Basic unit of a neural network that does simple calculations

Overfitting: When model memorizes training data instead of learning patterns

Payload: Malicious action performed by a trojan when triggered

RTL: Register-Transfer Level - abstraction level for hardware design

Tokenization: Breaking text into individual units (words/symbols)

Transformer: Architecture using self-attention, powers modern NLP

Trigger: Condition that activates a hardware trojan

Verilog: Hardware Description Language for designing circuits

Weight: Learnable parameter in a neural network

What's Next?

Congratulations! You now understand the basics of hardware trojan detection using deep learning. Here are some suggestions for continuing your learning: **1. Experiment with the Code:** • Try changing hyperparameters (learning rate, batch size) • Modify the network architecture (add/remove layers) • Test on your own Verilog files **2. Learn More About Deep Learning:** • Online courses: Coursera (Andrew Ng), Fast.ai • Books: "Deep Learning" by Goodfellow et al. • PyTorch tutorials: pytorch.org/tutorials **3. Explore Hardware Security:** • Research papers on hardware trojans • Trust-HUB benchmark datasets • IEEE Transactions on VLSI Systems **4. Contribute:** • Improve the detection models • Add new trojan types to the dataset • Create visualization tools **Good luck on your journey into AI and hardware security!**