Neural Vision

Interactive Machine Learning Visualizer

Built with Angular 21 + TensorFlow.js

A Modern TypeScript Application for Neural Network Education

Table of Contents

1. Project Objectives & Motivation
2. Analysis of Existing Solutions
3. Proposed Solution
4. Technical Architecture
5. Implementation Details
6. How to Use the Application
7. Results & Evaluation

Project Objectives

Why This Project?

Problem Statement:

- Neural networks are complex and difficult to understand
- Lack of interactive tools for learning ML concepts
- Existing solutions require installation and setup
- Need for real-time visualization during training

Target Audience:

- Computer Science students
- ML/AI beginners
- Educators teaching neural networks

Project Motivation

Educational Gap

Challenges in Learning Neural Networks:

- Abstract mathematical concepts
- Difficulty visualizing network architectures
- No immediate feedback during experimentation
- Limited understanding of layer functions

Our Goal:

Create an accessible, browser-based tool for hands-on ML learning

Analysis of Existing Solutions

Commercial Tools

TensorFlow Playground

- Simple and interactive
- Limited to simple networks
- No custom dataset support
- No model export

Netron

- Excellent visualization
- Read-only (no training)
- Requires model files
- No live interaction

Analysis of Existing Solutions (cont.)

Development Frameworks

Keras / TensorFlow

- Powerful and flexible
- Requires Python installation
- No built-in visualization
- Steep learning curve

PyTorch

- Research-friendly
- Complex for beginners
- No web interface
- Installation required

Technical Architecture

Technology Stack

Frontend Framework:

- Angular 21 (Standalone Components)
- TypeScript 5.9 (Strict Mode)

Styling:

- Tailwind CSS v4 (Utility-first)
- Custom CSS for animations

Machine Learning:

- TensorFlow.js (Browser ML)

Visualization:

- Three.js (3D Graphics)
- Chart.js (Metrics Graphs)

Backend (Development):

- JSON Server (REST API)

Technical Requirements

Functional Requirements

Must Have:

- Load datasets (MNIST, CSV, JSON)
- Build networks visually
- Train models in browser
- Display training metrics
- Make predictions

Nice to Have:

- 3D network visualization
- Export/import architectures
- Activation function graphs

Technical Requirements (cont.)

Non-Functional Requirements

Performance:

- Training in browser using WebGL
- Responsive UI (< 100ms interactions)
- Handle datasets up to 10,000 samples

Usability:

- Intuitive drag-and-drop interface
- Clear error messages
- Auto-validation of architectures

How to Use Neural Vision

Step 1: Load Dataset

Three Dataset Options:

1. **MNIST** - Click "Load MNIST" button
   - Pre-loaded handwritten digits
   - 60,000 training + 10,000 test samples
2. **CSV File** - Upload your data
   - Configure header row
   - Select label column
   - Set train/test split
3. **JSON** - Custom format support

How to Use (cont.)

Step 2: Build Network Architecture

Using the Builder:

1. Navigate to "Builder" tab
2. Drag layer types from palette:
    - Dense (fully connected)
    - Conv2D (convolutional)
    - MaxPooling
    - Dropout
    - Batch Normalization
3. Drop layers in canvas
4. Configure each layer (click to edit)
5. Set optimizer parameters

How to Use (cont.)

Step 3: Train the Model

Training Process:

1. Click "Build Neural Network"
2. Navigate to "Train" tab
3. Configure training:
    - Number of epochs
    - Batch size
    - Validation split
4. Click "Start Training"
5. Monitor real-time metrics:
    - Loss graph
    - Accuracy graph
    - Confusion matrix

How to Use (cont.)

Step 4: Visualize & Predict

Visualization:

- Navigate to "3D View" tab
- Explore network architecture in 3D
- See neuron connections

Making Predictions:

- Go to "Predict" tab
- Input test data
- View confidence scores
- See top predictions

Technical Architecture

Application Structure

6 Feature Components:

- Dataset Playground
- Network Builder
- Network Visualizer
- Training Dashboard
- Activation Visualizer
- Prediction Panel

4 Shared Services:

- Neural Network Service
- Dataset Service
- Training Service
- Model History Service

Components Overview

1. Dataset Playground

Purpose: Load and preview datasets

Features:

- File upload (MNIST, CSV, JSON)
- CSV parsing with auto-detection
- TensorFlow.js MNIST integration
- Live sample preview

Dataset Playground (Implementation)

Angular Concepts Used:

- **Signals** - Reactive state management
- **Template-driven forms** - Two-way binding with ngModel
- **Lifecycle hooks** - ngOnInit for initialization
- **Dependency Injection** - Service composition
- **Router** - Auto-navigation after load

Components Overview

2. Network Builder

Purpose: Visual architecture builder

Features:

- Drag-and-drop layer composition
- Real-time architecture validation
- Import/Export JSON architectures
- Dynamic form configuration

Network Builder (Implementation)

Angular Concepts Used:

- **Reactive Forms** - FormBuilder + Validators
- **Computed Signals** - isValid = computed(...)
- **Custom Directives** - Drag & drop functionality
- **FormGroups** - Layer & optimizer configs
- **Signal Updates** - .update() for arrays

Components (3/6)

3. Network Visualizer

Purpose: 3D visualization with Three.js

Features:

- Interactive 3D network graph
- Layer-by-layer exploration
- Neuron activation display

Angular Concepts:

- `@ViewChild` for DOM access
- `AfterViewInit` lifecycle hook
- ElementRef for WebGL canvas

Components (4/6)

4. Training Dashboard

Purpose: Real-time training metrics and charts

Features:

- Loss/Accuracy graphs (Chart.js)
- Confusion matrix calculation
- Precision, Recall, F1-Score
- Live epoch updates

Angular Concepts:

- Multiple `@ViewChild` decorators
- Async/await for training
- Callback functions for updates

Components (5/6)

5. Activation Visualizer

Purpose: Visualize activation functions

Features:

- ReLU, Sigmoid, Tanh, Softmax
- Mathematical formulas
- Interactive graphs

Angular Concepts:

- Property binding
- Event binding
- Pure functions

Components (6/6)

6. Prediction Panel

Purpose: Live predictions with trained model

Features:

- Real-time inference
- Confidence scores
- Top-N predictions

Angular Concepts:

- Async patterns
- Error handling
- Signal updates

Services (1/4)

Neural Network Service

Responsibility: TensorFlow.js model management

Key Methods:

- `buildModel()` - Create model from architecture
- `trainModel()` - Train with callbacks
- `predict()` - Make predictions
- `getLayerOutputs()` - Extract activations

Uses: Signals, RxJS, TensorFlow.js API

Services (2/4)

Dataset Service

Responsibility: Data loading and preprocessing

Features:

- MNIST loading from TensorFlow.js
- CSV parsing with type detection
- Automatic normalization
- Train/test splitting

Key Methods:

- `loadMNIST()`
- `loadFromCSV(file, config)`
- `getRandomSamples(split, count)`

Services (3/4)

Training Service

Responsibility: Training orchestration

Features:

- Metrics calculation
- Confusion matrix
- Validation handling

Services (4/4)

Model History Service

Responsibility: HTTP/REST API integration

```
@Injectable({ providedIn: 'root' })
export class ModelHistoryService {
  constructor(private http: HttpClient) {}

  getAllModels(): Observable<SavedModel[]>
  saveModel(model): Observable<SavedModel>
  updateModel(id, updates): Observable<SavedModel>
  deleteModel(id): Observable<void>
}
```

Backend: JSON Server (port 3001)

Custom Directives

1. Draggable Directive

```
@Directive({ selector: '[appDraggable]' })
```

- Makes layer templates draggable
- HTML5 Drag & Drop API

2. Drop Zone Directive

```
@Directive({ selector: '[appDropZone]' })
```

- Creates drop targets for layers

Custom Directives (cont.)

3. Digit Render Directive

```
@Directive({ selector: '[appDigitRender]' })
```

- Renders MNIST digits on canvas
- 28×28 pixel visualization

4. Neuron Highlight Directive

```
@Directive({ selector: '[appNeuronHighlight]' })
```

- Interactive neuron highlighting in 3D

Custom Pipes

Data Transformation

```
// activationName pipe
{{ 'relu' | activationName }}  // → "ReLU"

// numberFormat pipe
{{ 0.95432 | numberFormat:4 }}  // → "0.9543"

// percentage pipe
{{ 0.9543 | percentage }}  // → "95.43%"

// duration pipe
{{ 125000 | duration }}  // → "2m 5s"
```

TypeScript Models

Type Safety Throughout

```typescript
// Layer Configuration
export interface LayerConfig {
  id: string;
  type: LayerType;
  units?: number;
  activation?: ActivationFunction;
}

// Network Architecture
export interface NetworkArchitecture {
  name: string;
  layers: LayerConfig[];
  optimizer: OptimizerConfig;
}
```

Modern Angular Patterns

Standalone Components

```
@Component({
  selector: 'app-network-builder',
  standalone: true,
  imports: [CommonModule, FormsModule, ReactiveFormsModule],
  templateUrl: './network-builder.html',
})
export class NetworkBuilder { }
```

No NgModule Required!

Signals (Angular 16+)

Fine-Grained Reactivity

```
// Writable Signal
readonly layers = signal<LayerConfig[]>([]);

// Computed Signal
readonly isValid = computed(() =>
  this.validateArchitecture()
);

// Update Signal
this.layers.update(layers => [...layers, newLayer]);

// Read in Template
{{ layers().length }}
```

New Template Syntax (Angular 17+)

Modern Control Flow

```
<!-- Before: *ngIf -->
@if (layers().length === 0) {
  <div>No layers yet</div>
} @else {
  <div>{{ layers().length }} layers</div>
}

<!-- Before: *ngFor -->
@for (layer of layers(); track layer.id) {
  <div>{{ layer.name }}</div>
}
```

Cleaner, faster, type-safe!

Reactive Forms

Type-Safe Validation

~~~
layerConfigForm = this.fb.group({
  name: ['', Validators.required],
  units: [128, [Validators.min(1)]],
  activation: ['relu'],
  rate: [0.2, [Validators.min(0), Validators.max(0.99)]],
});

// Template
<form [formGroup]="layerConfigForm">
  <input formControlName="name" />
</form>

---

# Routing Configuration

```typescript
export const routes: Routes = [
  { path: '', redirectTo: 'dataset', pathMatch: 'full' },
  { path: 'dataset', component: DatasetPlayground },
  { path: 'builder', component: NetworkBuilder },
  { path: 'visualize', component: NetworkVisualizer },
  { path: 'train', component: TrainingDashboard },
  { path: 'activations', component: ActivationVisualizer },
  { path: 'predict', component: PredictionPanel },
];
~~~

Declarative routing with titles

Styling with Tailwind CSS

Tailwind CSS v4 Implementation

Where Tailwind is Used:

1. Global Styles (`styles.css`)

- CSS custom properties for color palette
- Utility classes for common patterns
- Responsive breakpoints

2. Component Templates

- Layout utilities (flex, grid)
- Spacing (padding, margin)
- Typography (font sizes, colors)
- Responsive design classes

Tailwind CSS (cont.)

Examples of Tailwind Usage

Layout Classes:

```
<div class="min-h-screen flex flex-col">
  <nav class="h-16 border-b sticky top-0">
  <main class="flex-1 relative overflow-hidden">
```

Component Styling:

```
<button class="px-5 py-2.5 rounded-xl
               bg-cyan-500 text-white
               hover:bg-cyan-600 transition-all">
```

Why Tailwind?

- Rapid UI development
- Consistent design system
- No CSS conflicts
- Responsive utilities built-in

🎨 Application Bootstrap

Functional Configuration (No NgModule)

```
// main.ts
bootstrapApplication(App, appConfig);

// app.config.ts
export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(routes),
    provideHttpClient(),
  ]
};
```

Dependency Injection

Service Composition

```
constructor(
  private fb: FormBuilder,
  private nnService: NeuralNetworkService,
  private datasetService: Dataset,
  private router: Router
) { }
```

Services marked with:

```
@Injectable({ providedIn: 'root' })
```

Testing Structure

Unit Tests Included

```
// *.spec.ts files for:
- Components
- Services
- Directives
```

Test Framework: Vitest

DOM Testing: JSDOM

Performance Features

Optimization Strategies

Signals - Better change detection
OnPush Strategy - Reduce rendering
Lazy Loading - Code splitting ready
Tree Shaking - Smaller bundles
Standalone Components - Optimal imports

Project Achievements

What We've Built

6 Complex Components with modern patterns
4 Injectable Services with clean architecture
4 Reusable Directives for UI interactions
4 Custom Pipes for data transformation
Full Type Safety with TypeScript strict mode
HTTP Integration with REST API
Form Validation with Reactive Forms
3D Visualization with Three.js
ML in Browser with TensorFlow.js

Application Workflow

```
1. Load Dataset (MNIST/CSV/JSON)
   ↓
2. Build Network Architecture (Drag & Drop)
   ↓
3. Configure Training Parameters
   ↓
4. Train Model (Real-time metrics)
   ↓
5. Visualize in 3D
   ↓
6. Make Predictions
```

🔥 Key Technical Highlights

Advanced Implementation

- **WebGL Rendering** for 3D network graph
- **Canvas API** for MNIST digit display
- **Web Workers** ready for heavy computations
- **Local Storage** for model persistence
- **IndexedDB** ready for large datasets
- **Service Workers** ready for offline mode

Build Configuration

Angular Configuration

```json
{
  "builder": "@angular/build:application",
  "options": {
    "browser": "src/main.ts",
    "tsConfig": "tsconfig.app.json"
  }
}
```

TypeScript: ES2022 target

Module: Bundler resolution

Strict Mode: Enabled

🌟 Design Patterns Used

Software Engineering Best Practices

Singleton Pattern - Services
Observer Pattern - RxJS Observables
Strategy Pattern - Optimizer configs
Factory Pattern - Model building
Dependency Injection - Loose coupling
Reactive Programming - Signals & Observables

Library Integration Details

Three.js - 3D Visualization

Where Used: `NetworkVisualizer` Component

Purpose: Render interactive 3D neural network graph

Implementation:

```
export class NetworkVisualizer implements AfterViewInit {
  private scene!: THREE.Scene;
  private camera!: THREE.PerspectiveCamera;
  private renderer!: THREE.WebGLRenderer;

  ngAfterViewInit(): void {
    this.initThreeJS();
    this.createNetworkMesh();
    this.animate();
  }
}
```

What it does:

- Creates 3D spheres for neurons
- Draws lines for connections
- Enables camera rotation & zoom

Library Integration (cont.)

Chart.js - Metrics Visualization

Where Used: `TrainingDashboard` Component

Purpose: Display real-time training metrics

Two Charts Created:

1. **Loss Chart** - Training & validation loss over epochs
2. **Accuracy Chart** - Training & validation accuracy

Code:

```
this.lossChart = new Chart(ctx, {
  type: 'line',
  data: { datasets: [trainLoss, valLoss] }
});
```

Library Integration (cont.)

TensorFlow.js - Machine Learning

Where Used: `NeuralNetworkService`

Purpose: Build, train, and run ML models in browser

Key Operations:

```
// Build model
this.model = tf.sequential({ layers });

// Train model
await this.model.fit(x, y, {
  epochs: 10,
  callbacks: { onEpochEnd: (...) => {...} }
});

// Make predictions
const result = this.model.predict(inputTensor);
```

Uses WebGL acceleration for performance

Production Quality

Enterprise-Ready Features

Error Handling - Try-catch throughout
Input Validation - Form validators
Type Safety - 100% TypeScript
Logging - Console logging system
Code Organization - Clean architecture
Separation of Concerns - Services vs Components
Reusability - Directives, Pipes, Services

Technology Integration

External Libraries

TensorFlow.js - ML models in browser
Three.js - 3D visualization
Chart.js - Metrics graphs
Tailwind CSS - Utility-first styling
JSON Server - Mock REST API

All seamlessly integrated with Angular!

Development Experience

Developer-Friendly Setup

```
# Install dependencies
npm install

# Start dev server + API
npm run dev

# Build for production
npm run build

# Run tests
npm test
```

Hot Module Replacement - Instant updates

TypeScript Checking - Real-time errors

Linting - Code quality enforcement

Responsive Design

Mobile-Friendly

- Gradient backgrounds
- Glass-morphism effects
- Cyberpunk theme (cyan/blue/purple)
- Smooth animations
- Custom scrollbars
- Hover effects

All styled with modern CSS!

Conclusion

What Makes This Special?

Modern Angular 21 - Latest features
Production Architecture - Scalable design
Type Safety - Strict TypeScript
Clean Code - Best practices
Full-Stack Ready - HTTP services
Machine Learning - TensorFlow.js
3D Graphics - Three.js integration
Comprehensive

Thank You! 🙏

Neural Vision

Interactive Machine Learning Visualizer

Questions?

Appendix: Commands Cheat Sheet

```
# Development
npm start            # Angular dev server
npm run dev          # Dev server + API
npm run api          # JSON Server only

# Building
npm run build        # Production build

# Testing
npm test            # Run unit tests
```

Appendix: File Structure

```
neural-vision/
├── src/
│   ├── app/
│   │   ├── components/     # 6 feature components
│   │   ├── services/       # 4 shared services
│   │   ├── directives/     # 4 custom directives
│   │   ├── pipes/          # 4 custom pipes
│   │   ├── models/         # TypeScript interfaces
│   │   └── app.config.ts   # App configuration
│   ├── styles.css          # Global styles
│   └── main.ts             # Bootstrap
├── angular.json            # Angular CLI config
├── tsconfig.json           # TypeScript config
└── package.json            # Dependencies
```