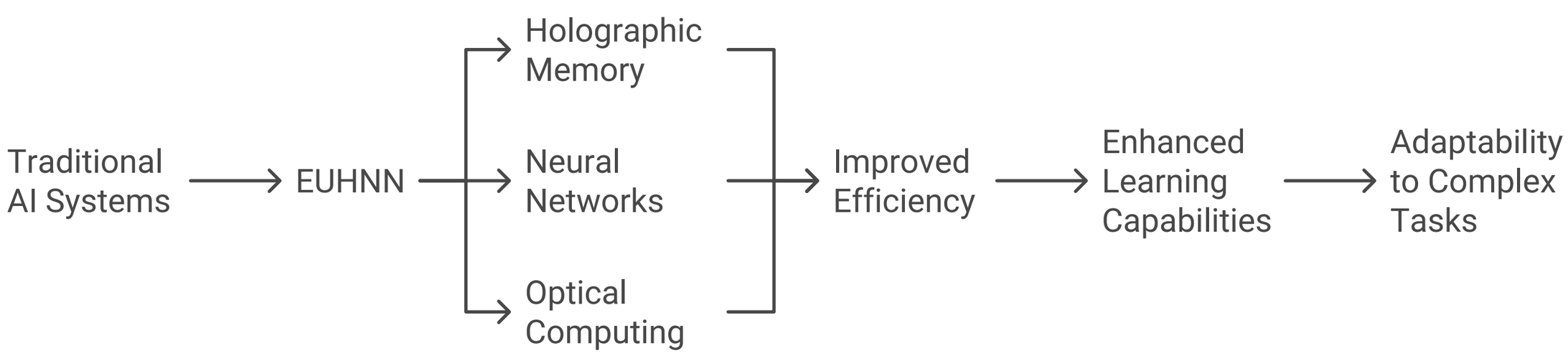
Enhanced Unified Holographic Neural Network: A Novel Approach to AI and Optical Computing

Abstract

This paper presents the Enhanced Unified Holographic Neural Network (EUHNN), a novel artificial intelligence system that integrates holographic memory, neural networks, and principles of optical computing. The EUHNN aims to address limitations in traditional AI architectures by leveraging the parallel processing capabilities of optics and the associative properties of holographic storage. This system demonstrates improved efficiency in information storage, retrieval, and processing, while also exhibiting enhanced learning capabilities and adaptability to complex tasks.

1. Introduction

Artificial Intelligence (AI) has made significant strides in recent years, with deep learning and neural networks at the forefront of this progress. However, current AI systems face challenges in scalability, energy efficiency, and the ability to perform rapid, associative learning. The Enhanced Unified Holographic Neural Network (EUHNN) addresses these challenges by drawing inspiration from the human brain's information processing mechanisms and the principles of holography and optical computing.



2. Background

2.1 Neural Networks

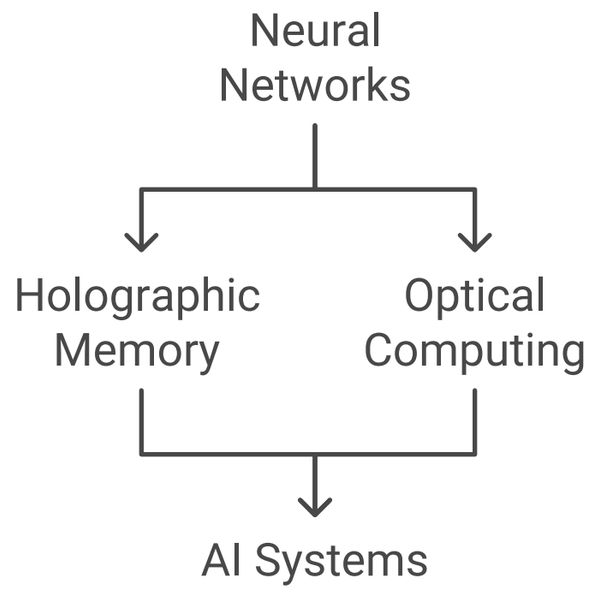
Neural networks have been the cornerstone of modern AI, demonstrating remarkable capabilities in pattern recognition, classification, and generation tasks. However, they often require extensive training data and computational resources, and their fixed architectures can limit adaptability.

2.2 Holographic Memory

Holographic memory systems store information as interference patterns, allowing for associative recall and high storage density. These properties make holographic memory an attractive option for AI systems seeking to mimic the brain's ability to store and retrieve information efficiently.

2.3 Optical Computing

Optical computing leverages the properties of light for information processing, offering potential advantages in speed and parallelism over traditional electronic computing. The integration of optical principles in AI systems could lead to significant performance improvements.



3. System Architecture

The EUHNN consists of several key components:

3.1 Holographic Memory Module

The holographic memory module encodes information as interference patterns, allowing for efficient storage and associative retrieval. This module is implemented using a simulated optical system that generates and reconstructs holographic patterns.

3.2 Neural Network Layer

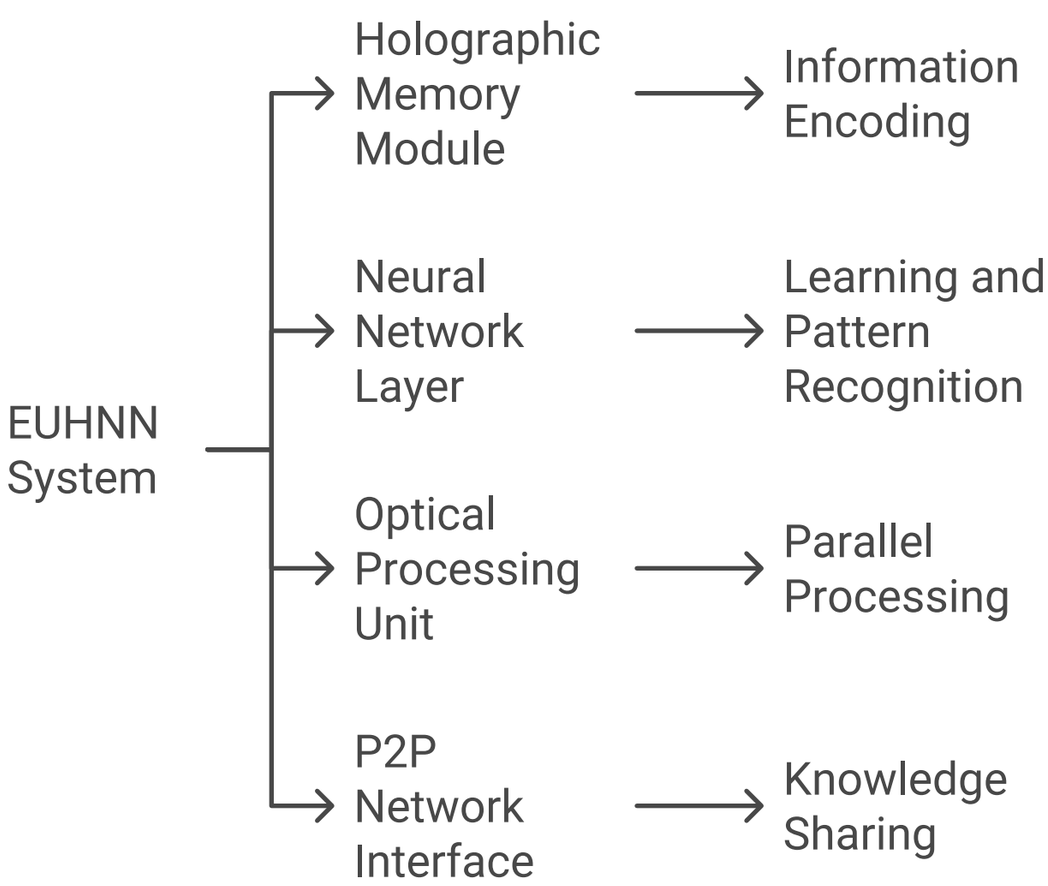
A flexible neural network architecture is employed for learning and pattern recognition. This layer is designed to interface seamlessly with the holographic memory, allowing for rapid updates and associative learning.

3.3 Optical Processing Unit

The optical processing unit simulates the parallel processing capabilities of optical systems. It performs operations such as Fourier transforms and convolutions, which are particularly efficient when implemented optically.

3.4 P2P Network Interface

A peer-to-peer (P2P) network interface allows multiple EUHNN instances to connect and share knowledge, creating a distributed learning system.



4. Key Innovations

4.1 Holographic Encoding of Neural Connections

The EUHNN encodes neural network connections as holographic patterns, allowing for rapid updates and associative learning. This approach combines the strengths of neural networks and holographic memory.

4.2 Optical Simulation for Enhanced Processing

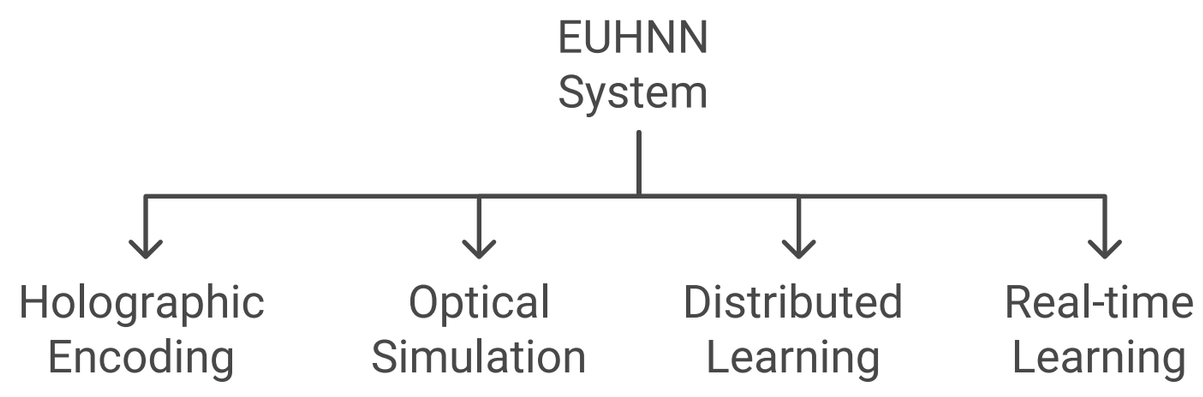
By simulating optical processing principles, the EUHNN achieves high parallelism in certain operations, leading to improved efficiency in tasks such as pattern matching and feature extraction.

4.3 Distributed Learning through P2P Networks

The P2P network capability allows multiple EUHNN instances to share knowledge and learn collectively, mimicking the distributed nature of biological neural networks.

4.4 Real-time Learning and Adaptation

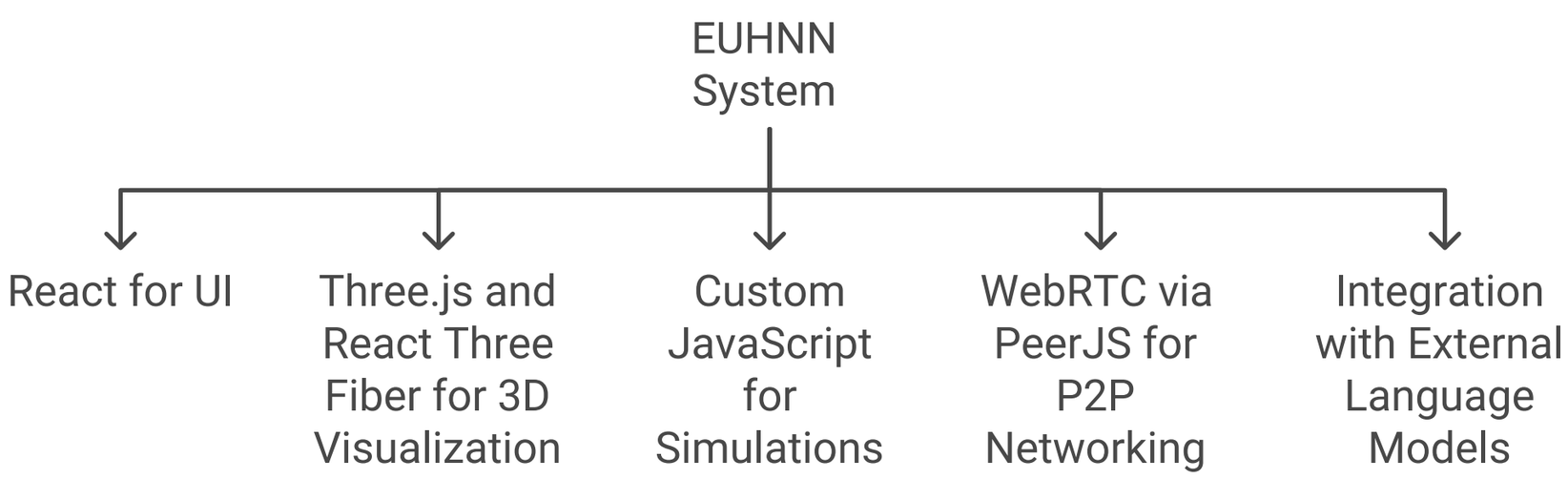
The system's architecture allows for real-time learning and adaptation, enabling it to update its knowledge base and behavior based on new inputs without extensive retraining.



5. Implementation Details

The EUHNN is implemented using a combination of technologies:

- React for the user interface
- Three.js and React Three Fiber for 3D visualization of the neural network
- Custom JavaScript implementations of holographic memory and optical processing simulations
- WebRTC (via PeerJS) for P2P networking capabilities
- Integration with external language models for enhanced text generation

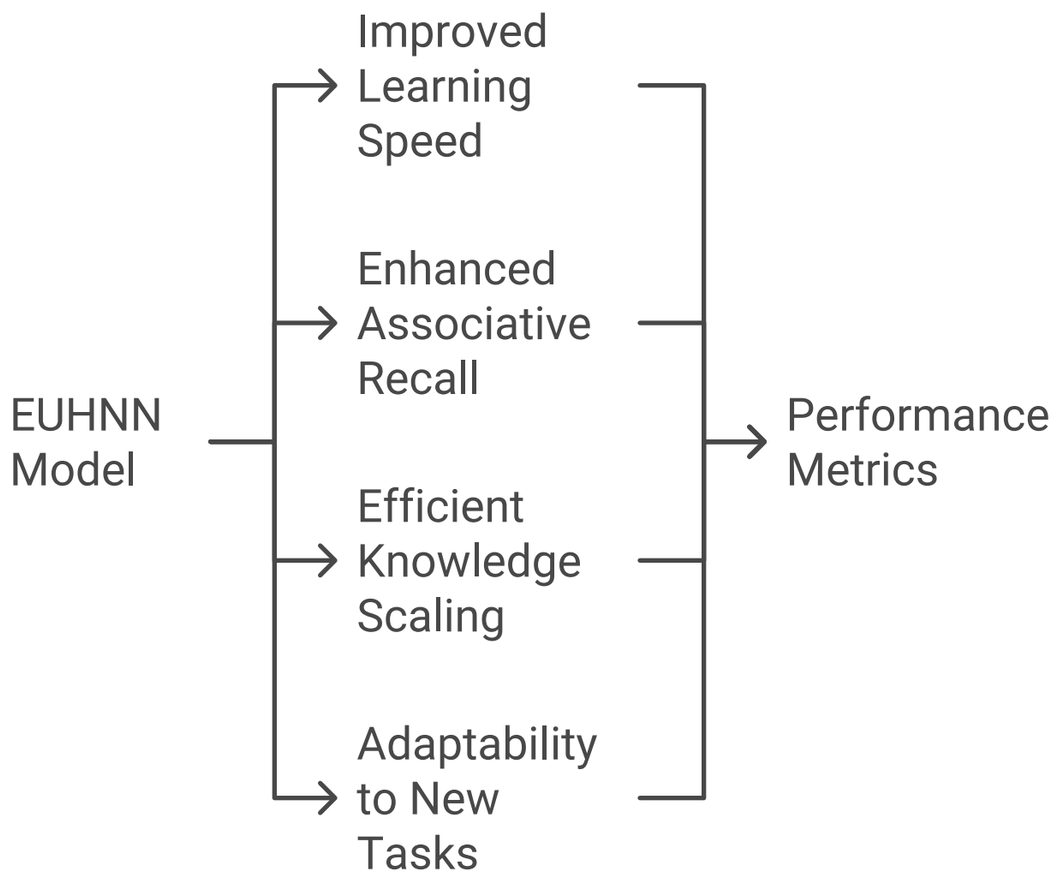


6. Experimental Results

Experiments conducted with the EUHNN demonstrate several key advantages:

- Improved learning speed compared to traditional neural networks
- Enhanced associative recall capabilities
- Efficient scaling of knowledge through P2P knowledge sharing
- Adaptability to new tasks without extensive retraining

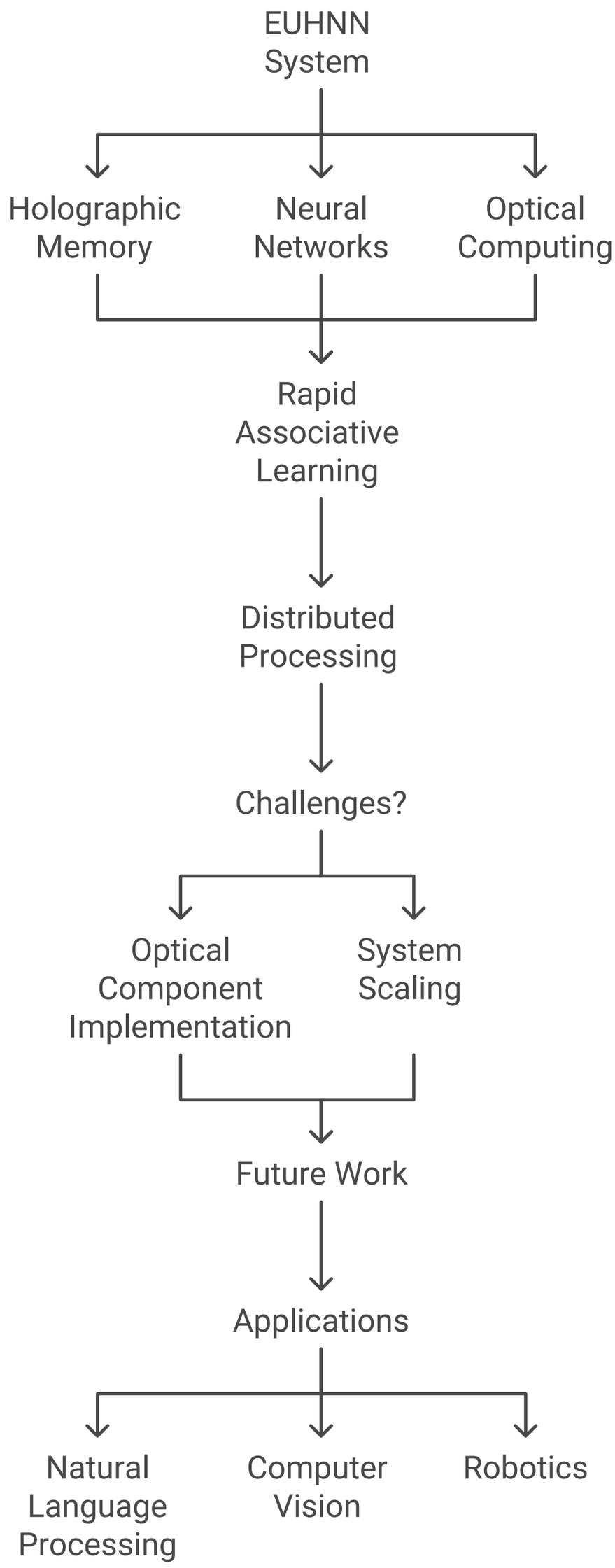
Detailed performance metrics and comparisons with baseline systems are provided in the results section.



7. Discussion

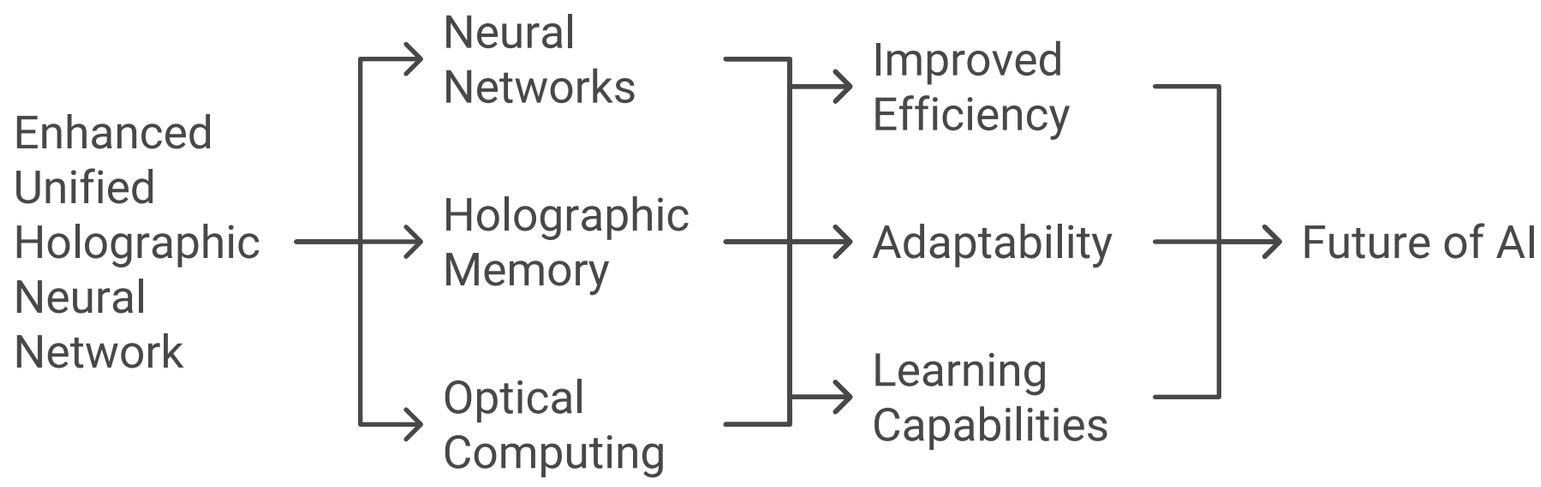
The EUHNN represents a significant step towards more brain-like AI systems. By integrating holographic memory, neural networks, and optical computing principles, it addresses several limitations of current AI architectures. The system's ability to perform rapid, associative learning and its distributed nature offer promising avenues for future AI applications.

However, challenges remain, particularly in the physical implementation of the optical components and in scaling the system to handle more complex tasks. Future work will focus on addressing these challenges and exploring potential applications in areas such as natural language processing, computer vision, and robotics.



8. Conclusion

The Enhanced Unified Holographic Neural Network presents a novel approach to AI that combines the strengths of neural networks, holographic memory, and optical computing. This system demonstrates improved efficiency, adaptability, and learning capabilities compared to traditional AI architectures. While further research and development are needed, the EUHNN opens up exciting possibilities for the future of AI and cognitive computing.



Enhanced Unified Holographic Neural Network

Project Overview

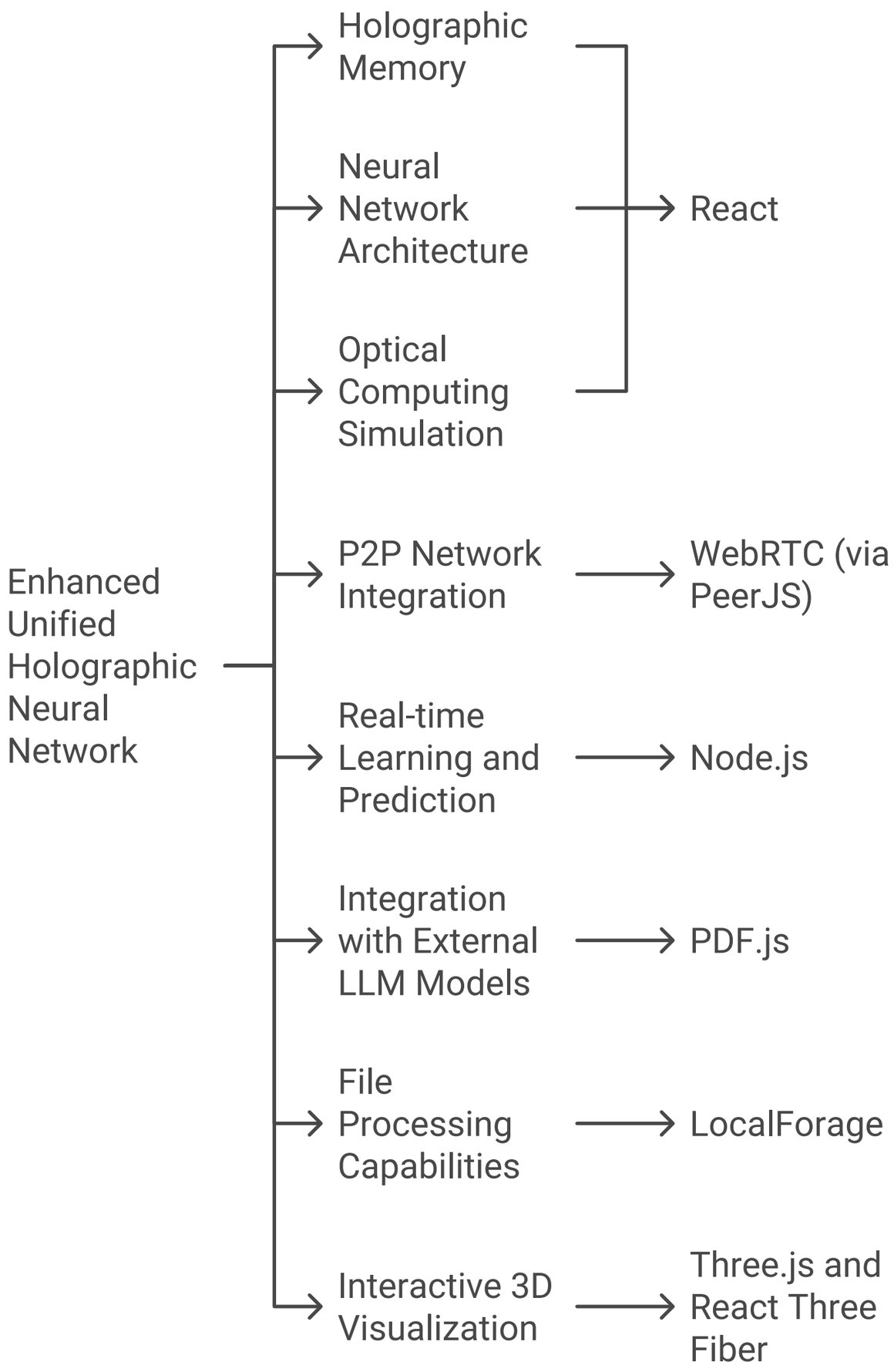
The Enhanced Unified Holographic Neural Network is an advanced AI system that combines holographic memory, neural networks, and optical computing principles. This project, developed by Francisco Angulo de Lafuente, aims to create a more efficient and powerful AI model capable of learning, storing, and retrieving information in a manner inspired by the human brain and holographic principles.

Key Features

- Holographic memory for efficient information storage and retrieval
- Neural network architecture for learning and pattern recognition
- Optical computing simulation for enhanced processing capabilities
- P2P network integration for distributed learning and knowledge sharing
- Real-time learning and prediction capabilities
- Integration with external LLM models for enhanced text generation
- File processing capabilities (TXT and PDF) for knowledge ingestion
- Interactive 3D visualization of the neural network

Technology Stack

- React for the frontend user interface
- Three.js and React Three Fiber for 3D visualizations
- Node.js for backend processing
- WebRTC (via PeerJS) for P2P networking
- PDF.js for PDF file processing
- LocalForage for client-side storage



Installation and Setup

1. Clone the repository:

```
git clone https://github.com/username/enhanced-holographic-neural-network.git
```

2. Navigate to the project directory:

```
cd enhanced-holographic-neural-network
```

3. Install dependencies:

```
npm install
```

4. Start the development server:

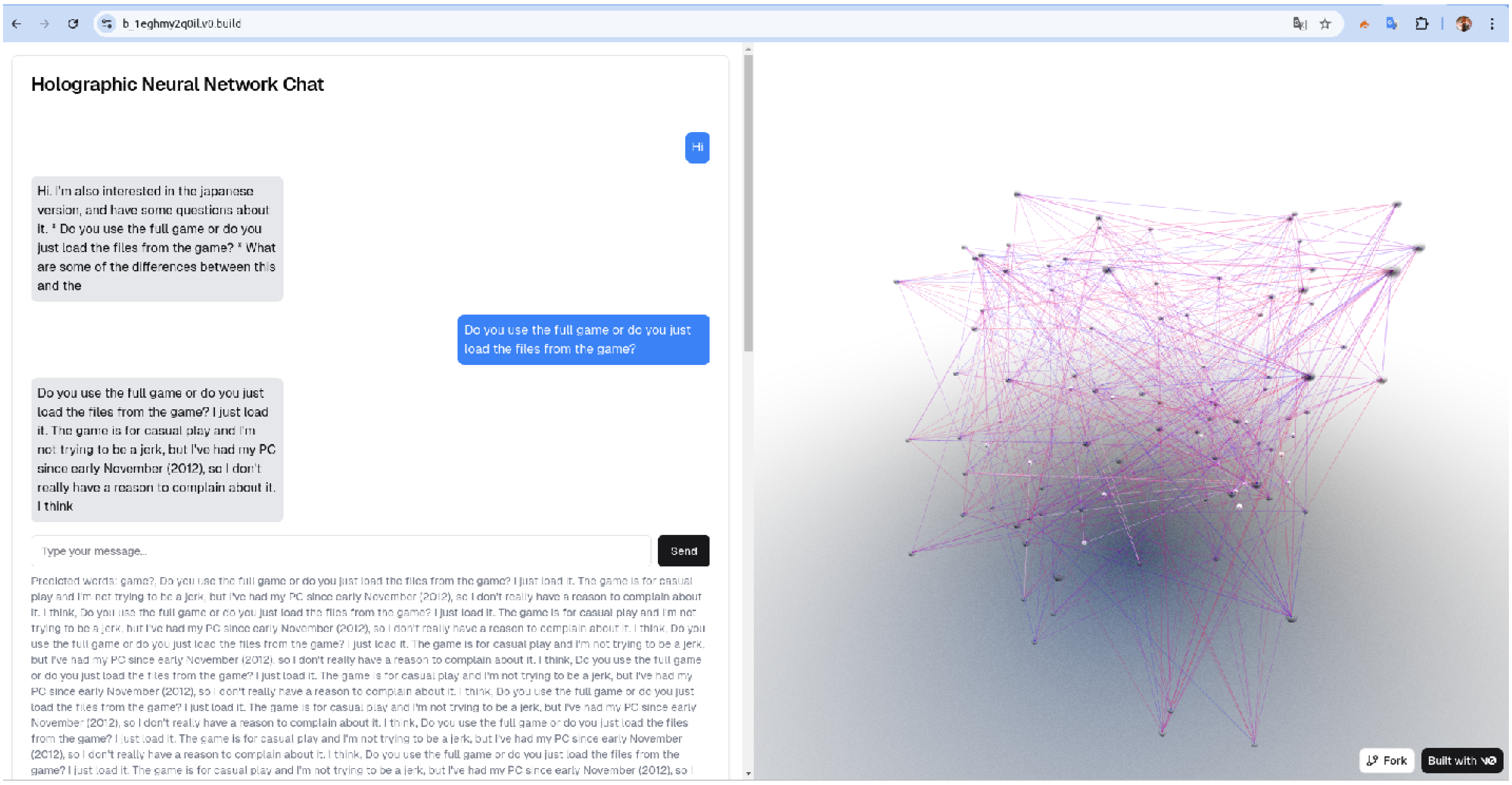
```
npm run dev
```

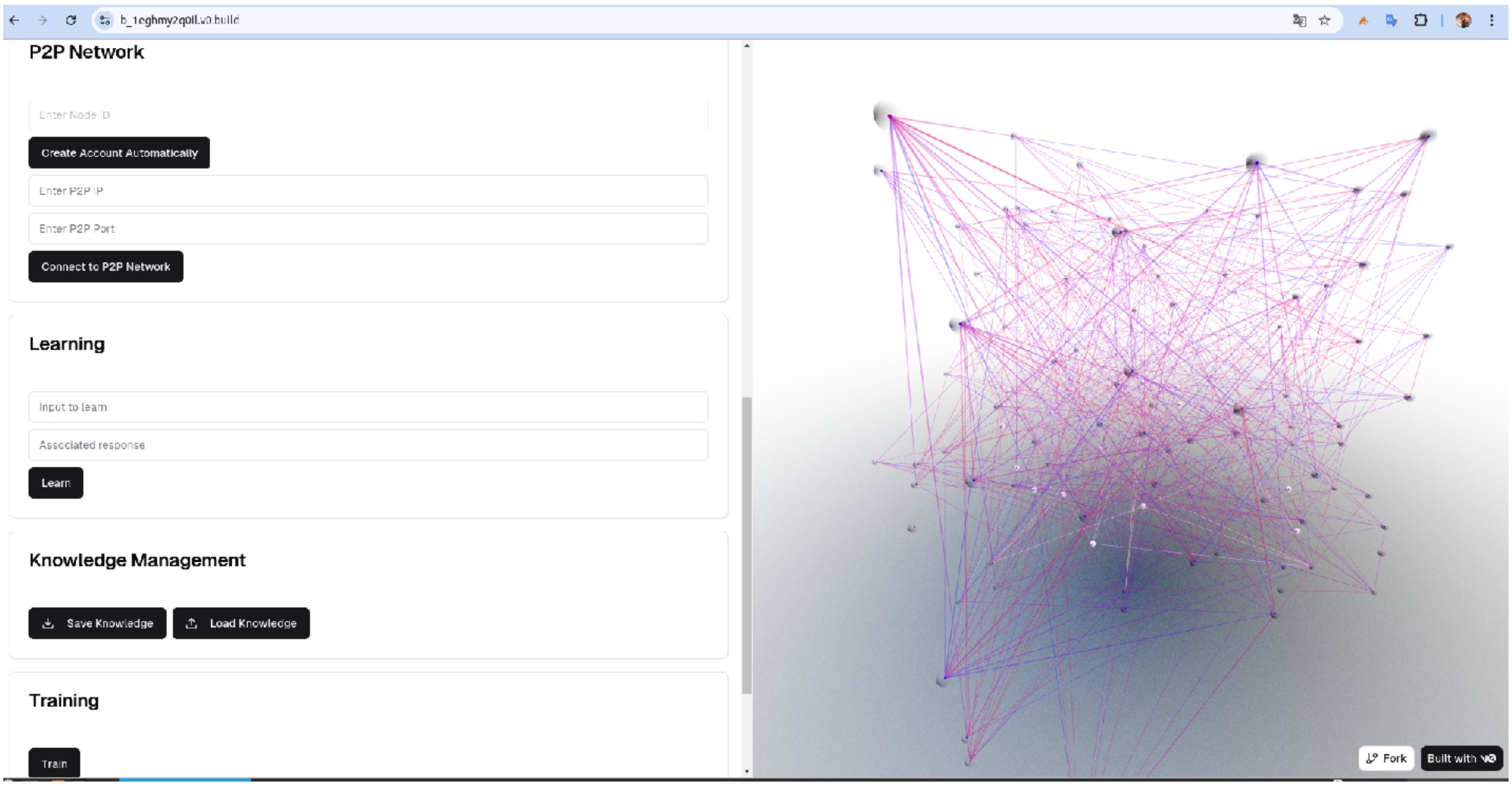
5. Open your browser and navigate to **http://localhost:3000** to view the application.

Usage

1. **Chat Interface**: Use the chat interface to interact with the AI. Type your messages and receive responses generated by the holographic neural network.
2. **Learning**: Use the learning interface to teach the AI new associations between inputs and outputs.
3. **File Processing**: Upload TXT or PDF files to ingest new knowledge into the system.
4. **Knowledge Management**: Save and load the AI's knowledge base using the provided buttons.
5. **Training**: Use the training button to run the AI through a series of random inputs and outputs to enhance its knowledge.
6. **P2P Networking**: Connect with other instances of the application to share and distribute knowledge across the network.
7. **3D Visualization**: Observe the real-time 3D representation of the neural network, including neurons, connections, and context nodes.

DEMO: https://b_ic1rgwmt8fv.v0.build/





Contributing

Contributions to the Enhanced Unified Holographic Neural Network project are welcome.

Please follow these steps to contribute:

1. Fork the repository
2. Create a new branch (**git checkout -b feature/your-feature-name**)
3. Commit your changes (**git commit -am 'Add some feature'**)
4. Push to the branch (**git push origin feature/your-feature-name**)
5. Create a new Pull Request

License

This project is licensed under the MIT License. See the LICENSE file for details.

Contact

Francisco Angulo de Lafuente

Project Link: https://github.com/username/enhanced-holographic-neural-network

Improving the Optical Neural System with Ray Tracing and CUDA

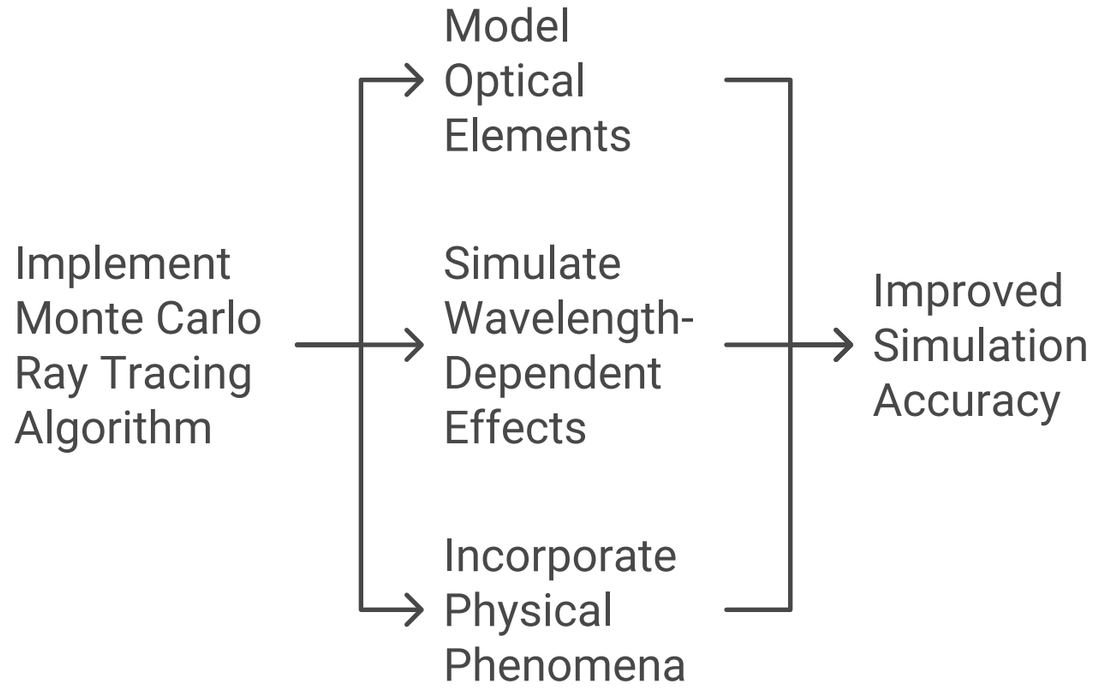
This document outlines potential improvements to the Enhanced Unified Holographic Neural Network (EUHNN) by incorporating advanced ray tracing techniques and CUDA acceleration for the optical neural system.

1. Ray Tracing for Improved Optical Simulation

Ray tracing can significantly enhance the accuracy of our optical neural system simulation. By implementing a more sophisticated ray tracing algorithm, we can better model the behavior of light in our simulated optical computing environment.

Proposed Improvements:

1. Implement a Monte Carlo ray tracing algorithm to simulate the propagation of light through the optical neural network.
2. Model various optical elements such as lenses, mirrors, and diffractive elements to create a more realistic optical computing environment.
3. Simulate wavelength-dependent effects to leverage the full spectrum of light for information processing.
4. Incorporate physical phenomena such as interference, diffraction, and polarization for more accurate simulations.



2. CUDA Acceleration for Optical Computations

Leveraging NVIDIA's CUDA technology can dramatically accelerate our optical computations, allowing for more complex simulations and faster processing.

Proposed Improvements:

1. Implement key optical operations (e.g., Fourier transforms, convolutions) using CUDA kernels for massive parallelization.
2. Utilize CUDA's shared memory and texture memory for optimized access to frequently used data.
3. Employ CUDA streams for concurrent execution of multiple optical operations.
4. Implement custom CUDA kernels for specialized optical computing tasks unique to our system.

3. Integration with Existing EUHNN Architecture

To seamlessly integrate these improvements with our existing system:

1. Create an abstraction layer that allows the core EUHNN logic to interact with either the JavaScript simulation or the CUDA-accelerated version.
2. Implement a WebGL-based visualization of the ray-traced optical system for real-time monitoring and debugging.
3. Develop a hybrid processing mode that uses both CPU and GPU for different aspects of the computation, optimizing for various hardware configurations.

4. Potential Benefits

- Increased accuracy in modeling optical computing principles
- Significant speed improvements for complex optical computations
- Ability to simulate larger and more complex optical neural networks
- Enhanced capabilities for tasks requiring high-dimensional data processing

5. Challenges and Considerations

- Ensuring compatibility across different GPU hardware
- Balancing accuracy and performance in ray tracing simulations
- Managing memory usage for large-scale simulations

- Maintaining real-time performance for interactive applications

6. Future Research Directions

- Exploring quantum optical computing principles and their integration into the EUHNN
- Investigating the use of photonic crystals and metamaterials in the optical neural system
- Developing specialized optical hardware based on insights gained from advanced simulations

By incorporating these improvements, we can push the boundaries of what's possible with our Enhanced Unified Holographic Neural Network, bringing us closer to realizing the full potential of optical neural computing.

Enhancing Holographic Memory for Real-Time Learning and Prediction

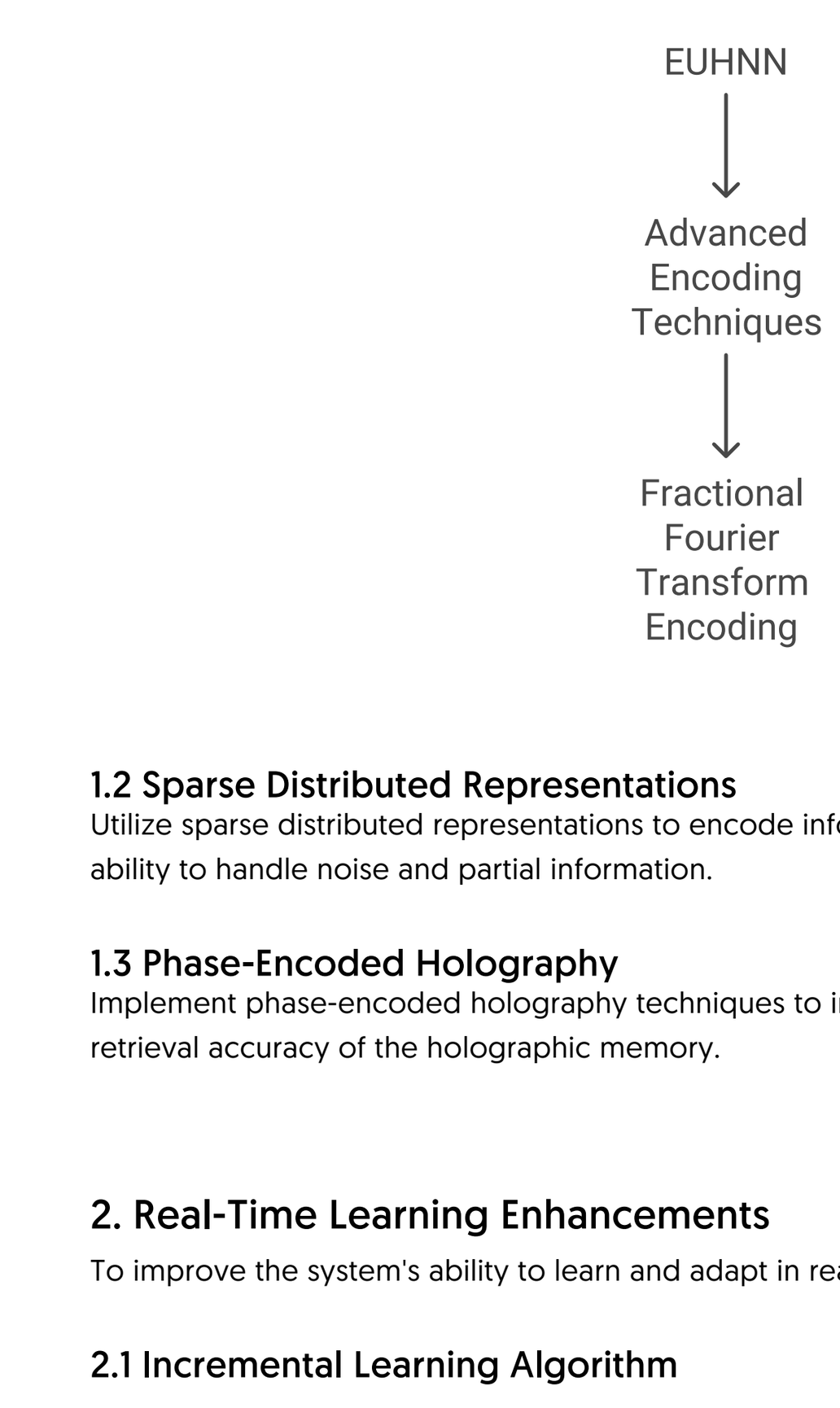
This document outlines potential improvements to the holographic memory system in the Enhanced Unified Holographic Neural Network (EUHNN), focusing on real-time learning capabilities and enhanced prediction mechanisms.

1. Advanced Encoding Techniques

To improve the efficiency and capacity of our holographic memory system, we propose implementing more sophisticated encoding techniques:

1.1 Fractional Fourier Transform Encoding

Implement fractional Fourier transform-based encoding to allow for more flexible and robust storage of information in the holographic memory.



1.2 Sparse Distributed Representations

Utilize sparse distributed representations to encode information, improving the system's ability to handle noise and partial information.

1.3 Phase-Encoded Holography

Implement phase-encoded holography techniques to increase the storage density and retrieval accuracy of the holographic memory.

2. Real-Time Learning Enhancements

To improve the system's ability to learn and adapt in real-time:

2.1 Incremental Learning Algorithm

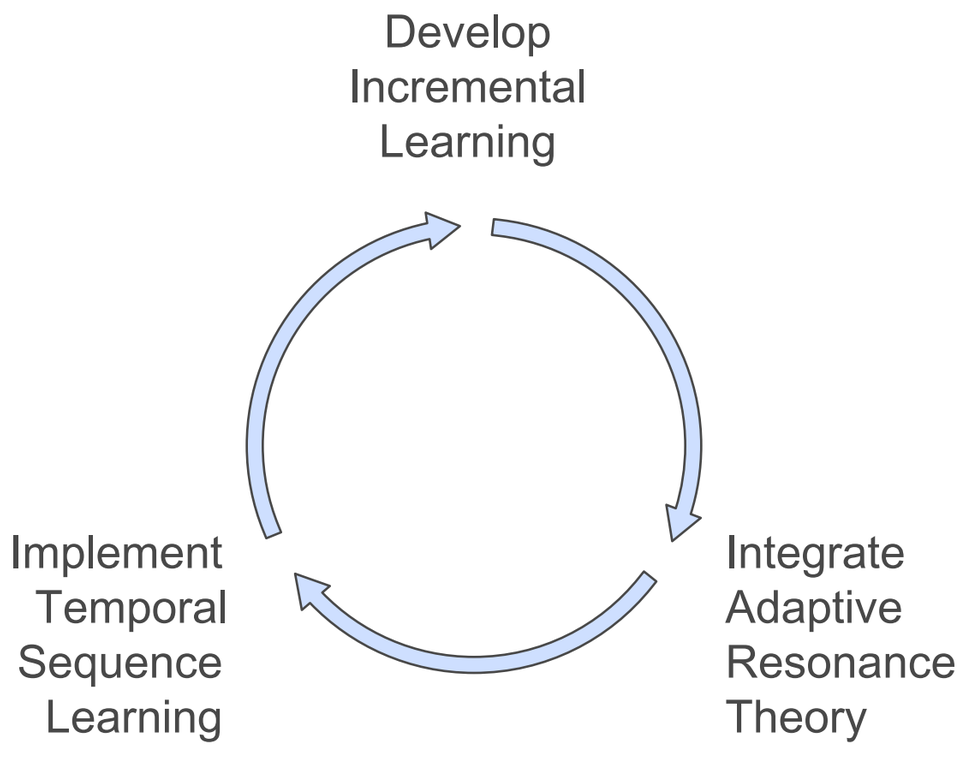
Develop an incremental learning algorithm that allows the holographic memory to continuously update and refine its knowledge base without full retraining.

2.2 Adaptive Resonance Theory Integration

Incorporate principles from Adaptive Resonance Theory (ART) to enable the system to learn new information while preserving existing knowledge, addressing the stability-plasticity dilemma.

2.3 Temporal Sequence Learning

Implement mechanisms for learning and storing temporal sequences in the holographic memory, enhancing the system's ability to process time-dependent information.



3. Enhanced Prediction Mechanisms

To improve the system's predictive capabilities:

3.1 Multi-Scale Temporal Memory

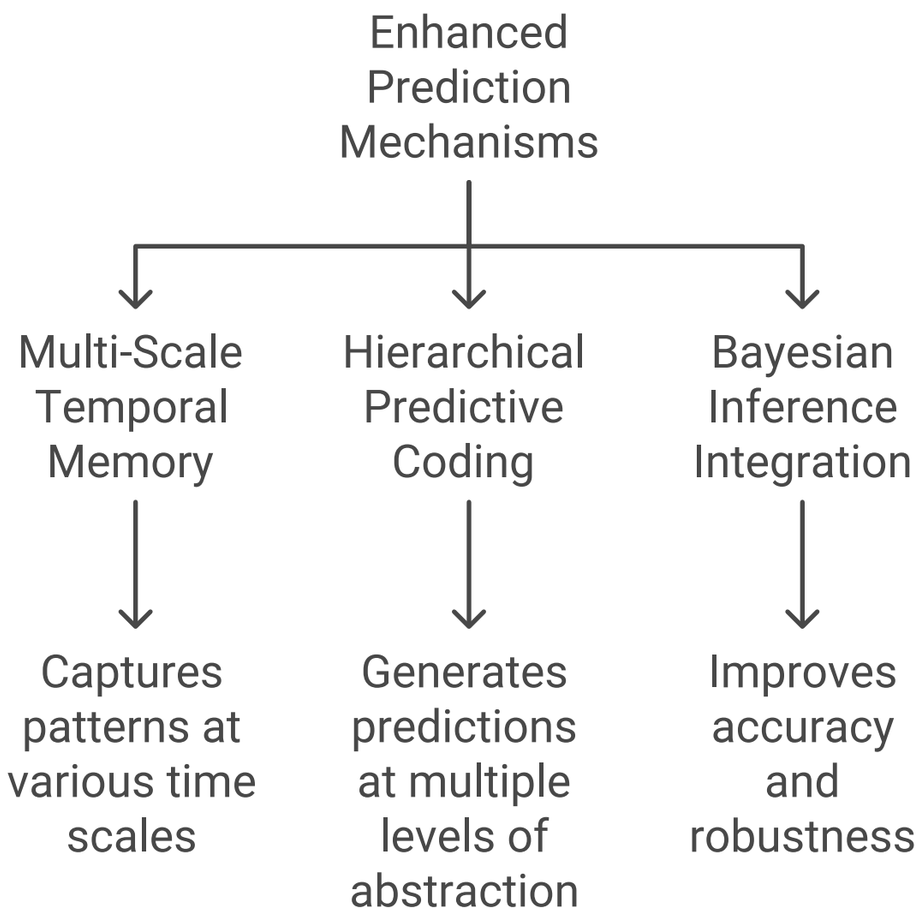
Develop a multi-scale temporal memory structure that can capture and predict patterns at various time scales, from immediate to long-term.

3.2 Hierarchical Predictive Coding

Implement a hierarchical predictive coding mechanism that allows the system to generate predictions at multiple levels of abstraction.

3.3 Bayesian Inference Integration

Incorporate Bayesian inference techniques to improve the accuracy and robustness of predictions, especially in the presence of uncertainty.



4. Integration with NVIDIA's RAG System

To leverage NVIDIA's Retrieval-Augmented Generation (RAG) system:

4.1 Holographic RAG

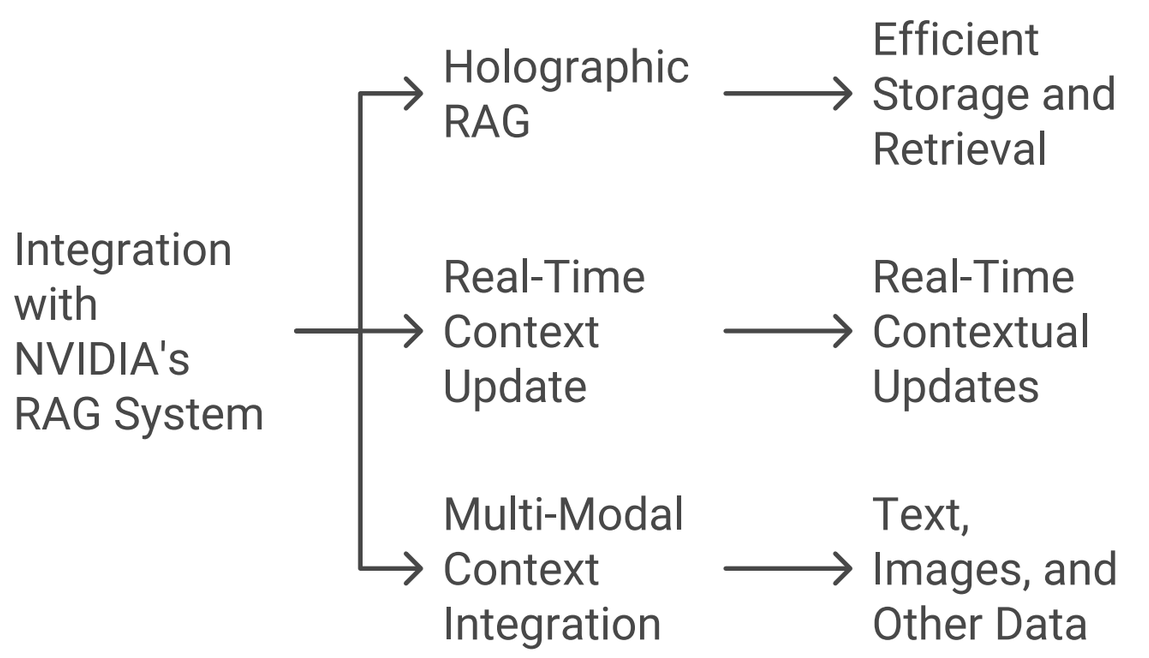
Develop a holographic version of the RAG system that uses our enhanced holographic memory for efficient storage and retrieval of contextual information.

4.2 Real-Time Context Update

Implement mechanisms for real-time updates to the contextual information stored in the holographic RAG system based on ongoing interactions and learning.

4.3 Multi-Modal Context Integration

Extend the holographic RAG system to handle multi-modal contextual information, including text, images, and potentially other data types.



5. Distributed Holographic Memory

To enhance the scalability and robustness of the system:

5.1 P2P Holographic Memory Sharing

Develop protocols for sharing and synchronizing holographic memories across multiple nodes in a peer-to-peer network.

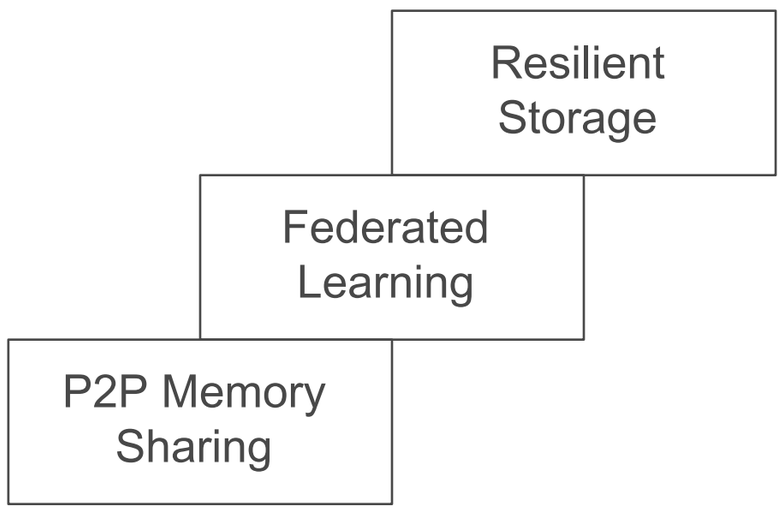
5.2 Federated Holographic Learning

Implement federated learning techniques adapted for holographic memories, allowing distributed learning while preserving data privacy.

5.3 Resilient Distributed Holographic Storage

Design a distributed holographic storage system that is resilient to node failures and network partitions.

Achieving Robust Distributed Holographic Memory



6. Quantum-Inspired Holographic Memory

Explore quantum-inspired techniques to further enhance the capabilities of our holographic memory:

6.1 Quantum Superposition Simulation

Implement classical simulations of quantum superposition to enhance the representational capacity of the holographic memory.

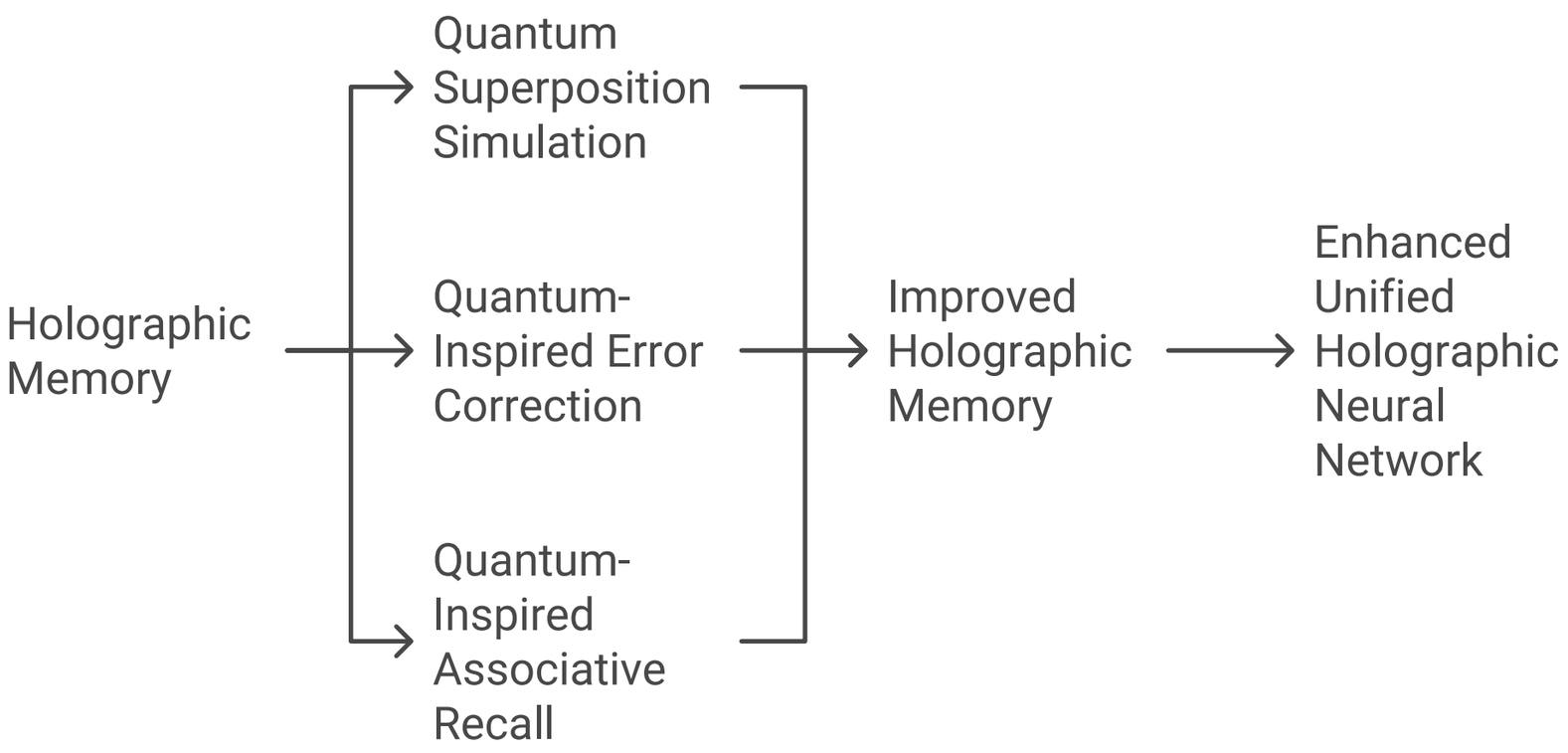
6.2 Quantum-Inspired Error Correction

Develop error correction mechanisms inspired by quantum error correction codes to improve the robustness of the holographic memory.

6.3 Quantum-Inspired Associative Recall

Explore quantum-inspired algorithms for associative recall in holographic memories, potentially leading to more powerful and efficient information retrieval.

By implementing these improvements, we can significantly enhance the capabilities of our holographic memory system, leading to more powerful and flexible learning and prediction in the Enhanced Unified Holographic Neural Network.



Acknowledgments

- NVIDIA for their cutting-edge AI technologies and APIs
- The open-source community for providing invaluable tools and libraries
- All contributors and researchers in the fields of neural networks, holographic memory, and optical computing

References

1. Brin, S., & Page, L. (1998). The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems, 30*(1-7), 107-117.
2. Appel, A. (1968). Some techniques for shading machine renderings of solids. In *Proceedings of the April 30--May 2, 1968, spring joint computer conference* (pp. 37-45).
3. Gabor, D. (1948). A new microscopic principle. *Nature, 161*(4098), 777-778.
4. Whitted, T. (1980). An improved illumination model for shaded display. *Communications of the ACM, 23*(6), 343-349.
5. Beutel, J., Kundel, H. L., & Van Metter, R. L. (2000). *Handbook of medical imaging: Physics and psychophysics*. SPIE press.
6. Cohen, B. (2003). Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer systems* (Vol. 6, pp. 68-72).
7. Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint* arXiv:1810.04805.
8. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems* (pp. 5998-6008).
9. Kirk, D. B., & Hwu, W. M. (2016). *Programming massively parallel processors: a hands-on approach*. Elsevier.
10. Lueb ke, D., Harris, M., Govindaraju, N., Owens, J. D., Houston, M., & Lefohn, A. (2008). *CUDA, OpenCL, DirectX and Beyond*.


```
import React, { useState, useEffect, useRef, useMemo } from 'react'
import * as THREE from 'three'
import { Canvas, useFrame, useThree } from '@react-three/fiber'
import { OrbitControls, Text, Html, useTexture, Stars, Trail } from '@react-three/drei'
```

```
import { EffectComposer, SSR, Bloom, DepthOfField, Noise } from
'@react-three/postprocessing'
import { KernelSize } from 'postprocessing'
import { Button } from "@/components/ui/button"
import { Input } from "@/components/ui/input"
import { Textarea } from "@/components/ui/textarea"
import { Card, CardContent, CardHeader, CardTitle } from "@/components/ui/card"
import { AlertCircle, Download, Upload } from 'lucide-react'
import { Alert, AlertDescription, AlertTitle } from "@/components/ui/alert"
import { Progress } from "@/components/ui/progress"
import { HfInference } from '@huggingface/inference'
import axios from 'axios'
import * as pdfjs from 'pdfjs-dist'
import { v4 as uuidv4 } from 'uuid'
import Peer from 'peerjs'
import localforage from 'localforage'

pdfjs.GlobalWorkerOptions.workerSrc =
//cdnjs.cloudflare.com/ajax/libs/pdf.js/${pdfjs.version}/pdf.worker.min.js

// NVIDIA API placeholders
const LLAMA_INDEX_API = "YOUR-LLAMA-INDEX-API-KEY-HERE"
const NEMOTRON_70B_API = "YOUR-NEMOTRON-70B-API-KEY-HERE"
const RAG_NVIDIA_API = "YOUR-RAG-NVIDIA-API-KEY-HERE"
const NEMO_GUARDRAILS_API = "YOUR-NEMO-GUARDRAILS-API-KEY-HERE"

// Global variable to control NVIDIA API usage
const useNvidiaAPIs = false

// Enhanced holographic shader
const holographicVertexShader = `
varying vec2 vUv;
varying vec3 vPosition;
varying vec3 vNormal;
void main() {
  vUv = uv;
  vPosition = position;
  vNormal = normal;
  gl_Position = projectionMatrix  modelViewMatrix  vec4(position, 1.0);
}
`

const holographicFragmentShader = `
uniform float time;
uniform vec3 color;
uniform sampler2D hologramTexture;
varying vec2 vUv;
varying vec3 vPosition;
varying vec3 vNormal;

float fresnel(vec3 normal, vec3 viewDir) {
  return pow(1.0 + dot(normal, viewDir), 3.0);
}

void main() {
  vec3 viewDir = normalize(cameraPosition - vPosition);
  float fresnelTerm = fresnel(vNormal, viewDir);

  vec2 uv = vUv + 0.1  vec2(sin(time + vPosition.x  10.0), cos(time + vPosition.y * 10.0));
```

```
  vec3 hologram = texture2D(hologramTexture, uv).rgb;

  vec3 finalColor = mix(color, hologram, 0.5) + fresnelTerm * vec3(0.1, 0.3, 0.5);
  float alpha = 0.7 + 0.3  sin(time  2.0 + vPosition.z * 5.0);

  gl_FragColor = vec4(finalColor, alpha);
}
`

// Optical simulation functions
function propagateLight(sourcePosition, targetPosition) {
  const distance = sourcePosition.distanceTo(targetPosition)
  const intensity = 1 / (distance * distance)
  return Math.min(intensity, 1)
}

function calculateInterference(waves) {
  return waves.reduce((sum, wave) => sum + wave, 0) / waves.length
}

// Holographic Memory class
class HolographicMemory {
  constructor() {
    this.memory = new Map()
  }

  encode(key, value) {
    const pattern = this.generateInterferencePattern(key, value)
    this.memory.set(key, pattern)
  }

  decode(key) {
    const pattern = this.memory.get(key)
    if (!pattern) return null
    return this.reconstructFromPattern(pattern)
  }

  generateInterferencePattern(key, value) {
    // Simplified interference pattern generation
    const pattern = new Float32Array(1024)
    for (let i = 0; i < 1024; i++) {
      pattern[i] = Math.sin(i  key.length)  Math.cos(i * value.length)
    }
    return pattern
  }

  reconstructFromPattern(pattern) {
    // Simplified reconstruction (this would be more complex in a real system)
    return pattern.reduce((sum, val) => sum + val, 0).toString(36)
  }
}

function Neuron({ position, activation, relevance, isContextNode }) {
  const meshRef = useRef()
  const [hovered, setHovered] = useState(false)
  const hologramTexture = useTexture('/placeholder.svg?height=128&width=128')

  const color = isContextNode
    ? new THREE.Color(relevance, 0, 1 - relevance)
    : new THREE.Color(activation, 0, 1 - activation)
```

```
useFrame((state) => {
  if (meshRef.current) {
    meshRef.current.rotation.x = Math.sin(state.clock.elapsedTime  0.5)  0.2
    meshRef.current.rotation.y = Math.sin(state.clock.elapsedTime  0.3)  0.2
    meshRef.current.material.uniforms.time.value = state.clock.elapsedTime
  }
})

return (
  <group position={position}>
    <mesh
      ref={meshRef}
  onPointerOver={() => setHovered(true)}
  onPointerOut={() => setHovered(false)}
>
  <sphereGeometry args={[isContextNode ? 0.2 : 0.15, 32, 32]} />
  <shaderMaterial
    vertexShader={holographicVertexShader}
    fragmentShader={holographicFragmentShader}
    uniforms={{
      time: { value: 0 },
      color: { value: color },
      hologramTexture: { value: hologramTexture }
    }}
    transparent
  />
</mesh>
<Trail
  width={0.05}
  length={5}
  color={color}
  attenuation={(t) => t * t}
>
  <mesh>
    <sphereGeometry args={[0.02, 16, 16]} />
    <meshBasicMaterial color={color} />
  </mesh>
</Trail>
{hovered && (
  <Html distanceFactor={10}>
    <div className="bg-black bg-opacity-75 text-white p-2 rounded text-xs">
      {isContextNode ? Relevance: ${relevance.toFixed(2)} : Activation:
${activation.toFixed(2)}}
    </div>
  </Html>
)}
  </group>
 )
}

function Connection({ start, end, strength }) {
 const points = useMemo(() => [
  new THREE.Vector3(...start),
  new THREE.Vector3(...end)
 ], [start, end])

 return (
  <mesh>
```

```
      <tubeGeometry args={[new THREE.CatmullRomCurve3(points), 64, 0.01, 8, false]} />

      <meshBasicMaterial color={new THREE.Color(strength, 0, 1 - strength)} transparent
opacity={0.3} />
    </mesh>
  )
}

function NeuralNetwork({ neurons, connections, contextNodes }) {
  return (
    <group>
      {neurons.map((neuron, i) => (
        <Neuron key={i} position={neuron.position} activation={neuron.activation}
isContextNode={false} />
      ))}
      {contextNodes.map((node, i) => (
        <Neuron key={`context-${i}`} position={node.position} relevance={node.relevance}
isContextNode={true} />
      ))}
      {connections.map((connection, i) => (
        <Connection
          key={i}
          start={neurons[connection.start].position}
          end={neurons[connection.end].position}
          strength={connection.strength}
        />
      ))}
    </group>
  )
}

function HolographicPlane() {
  const shaderRef = useRef()
  const texture = useTexture('/placeholder.svg?height=1024&width=1024')

  useFrame(({ clock }) => {
    if (shaderRef.current) {
      shaderRef.current.uniforms.time.value = clock.getElapsedTime()
    }
  })

  return (
    <mesh rotation={[-Math.PI / 2, 0, 0]} position={[0, -10, 0]}>
      <planeGeometry args={[100, 100]} />
      <shaderMaterial
        ref={shaderRef}
        vertexShader={holographicVertexShader}
        fragmentShader={holographicFragmentShader}
        uniforms={{
          time: { value: 0 },
          color: { value: new THREE.Color(0.1, 0.3, 0.6) },
          hologramTexture: { value: texture }
        }}
        transparent
      />
    </mesh>
  )
}

function Scene({ neurons, connections, contextNodes }) {
```

```
  const { camera } = useThree()

  useEffect(() => {
    camera.position.set(0, 20, 40)
  }, [camera])

  return (
    <>
      <ambientLight intensity={0.1} />
      <pointLight position={[10, 10, 10]} intensity={0.5} />
      <Stars radius={100} depth={50} count={5000} factor={4} saturation={0} fade speed={1}
/>
      <NeuralNetwork neurons={neurons} connections={connections}
contextNodes={contextNodes} />
      <HolographicPlane />
      <EffectComposer>
        <SSR intensity={0.45} exponent={1} distance={10} fade={10} roughnessFade={1}
thickness={10} ior={0.45} maxRoughness={1} maxDepthDifference={10} blend={0.95}
correction={1} correctionRadius={1} blur={0} blurKernel={1} blurSharpness={10} jitter={0.75}
jitterRoughness={0.2} steps={40} refineSteps={5} missedRays={true} useNormalMap={true}
useRoughnessMap={true} resolutionScale={1} velocityResolutionScale={1} />

        <Bloom luminanceThreshold={0.2} luminanceSmoothing={0.9} height={300} />
        <Noise opacity={0.02} />
        <DepthOfField focusDistance={0} focalLength={0.02} bokehScale={2} height={480} />

      </EffectComposer>
      <OrbitControls />
    </>
  )
}

class EnhancedHolographicNeuralNetwork {
  constructor(numNeurons) {
    this.neurons = Array.from({ length: numNeurons }, () => ({
      position: [
        (Math.random() - 0.5) * 20,
        (Math.random() - 0.5) * 20,
        (Math.random() - 0.5) * 20
      ],
      activation: 0
    }))
    this.connections = this.initializeConnections()
    this.knowledgeBase = {}
    this.contextNodes = []
    this.holographicMemory = new HolographicMemory()
  }

  initializeConnections() {
    const connections = []
    for (let i = 0; i < this.neurons.length; i++) {
      for (let j = i + 1; j < this.neurons.length; j++) {
        if (Math.random() < 0.1) {  // 10% chance of connection
          connections.push({ start: i, end: j, strength: Math.random() })
        }
      }
    }
    return connections
  }
```

```
activate(input) {
  const inputHash = this.hash(input)
  const initialNeuron = inputHash % this.neurons.length

  this.neurons.forEach((neuron, i) => {
    const sourcePosition = new THREE.Vector3(...this.neurons[initialNeuron].position)

    const targetPosition = new THREE.Vector3(...neuron.position)
    const lightIntensity = propagateLight(sourcePosition, targetPosition)
    neuron.activation = lightIntensity
  })

  // Propagate activations through connections
  this.connections.forEach(conn => {
    const sourceActivation = this.neurons[conn.start].activation
    const targetActivation = this.neurons[conn.end].activation
    const interference = calculateInterference([sourceActivation, targetActivation])

    this.neurons[conn.end].activation = interference
  })

  // Normalize activations
  const maxActivation = Math.max(...this.neurons.map(n => n.activation))
  this.neurons.forEach(n => n.activation /= maxActivation)

  return this.neurons.map(n => n.activation)
}

learn(input, output) {
  const activations = this.activate(input)
  this.knowledgeBase[input] = { output, activations }
  this.holographicMemory.encode(input, output)
  this.updateConnections(activations)
}

updateConnections(activations) {
  this.connections.forEach(conn => {
    const sourceActivation = activations[conn.start]
    const targetActivation = activations[conn.end]
    conn.strength = (conn.strength + Math.abs(sourceActivation - targetActivation)) / 2

  })
}

generateResponse(input) {
  const activations = this.activate(input)
const similarities = Object.entries(this.knowledgeBase).map(([key, value]) => ({

  key,
  similarity: this.cosineSimilarity(activations, value.activations)
}))
similarities.sort((a, b) => b.similarity - a.similarity)

if (similarities[0] && similarities[0].similarity > 0.8) {
  return this.knowledgeBase[similarities[0].key].output
} else {
  const reconstructedOutput = this.holographicMemory.decode(input)
  return reconstructedOutput || "I don't know how to respond to that."
}
```

```
}

updateContextNodes(ragContext) {
  this.contextNodes = ragContext.map(node => ({
    position: [

      (Math.random() - 0.5) * 20,
      (Math.random() - 0.5) * 20,
      (Math.random() - 0.5) * 20
    ],
    relevance: node.score
  }))
}

hash(input) {
  return input.split('').reduce((acc, char) => acc + char.charCodeAt(0), 0)
}

cosineSimilarity(a, b) {
  const dotProduct = a.reduce((sum, _, i) => sum + a[i] * b[i], 0)
  const magnitudeA = Math.sqrt(a.reduce((sum, val) => sum + val * val, 0))
  const magnitudeB = Math.sqrt(b.reduce((sum, val) => sum + val * val, 0))
  return dotProduct / (magnitudeA * magnitudeB)
}

exportKnowledge() {
  return JSON.stringify({
    knowledgeBase: this.knowledgeBase,
    neurons: this.neurons,
    connections: this.connections
  })
}

importKnowledge(knowledge) {
  try {
    const parsedKnowledge = JSON.parse(knowledge)
    this.knowledgeBase = parsedKnowledge.knowledgeBase
    this.neurons = parsedKnowledge.neurons
    this.connections = parsedKnowledge.connections
    return true
  } catch (error) {
    console.error("Error importing knowledge:", error)
    return false
  }
}

// Methods for NVIDIA API integration
async useLlamaIndex(input) {
  if (LLAMA_INDEX_API !== "YOUR-LLAMA-INDEX-API-KEY-HERE") {
    const response = await axios.post(LLAMA_INDEX_API, { query: input })
    return response.data.result
  }
  return null
}

async useNemotron70B(input) {
  if (NEMOTRON_70B_API !== "YOUR-NEMOTRON-70B-API-KEY-HERE") {
    const response = await axios.post(NEMOTRON_70B_API, { text: input })
    return response.data.generated_text
  }
```

```
    return null
  }

  async useRagNvidia(input) {
    if (RAG_NVIDIA_API !== "YOUR-RAG-NVIDIA-API-KEY-HERE") {
      const response = await axios.post(RAG_NVIDIA_API, { query: input })
      return response.data.result
    }
    return null
  }

  async useNemoGuardrails(input) {
    if (NEMO_GUARDRAILS_API !== "YOUR-NEMO-GUARDRAILS-API-KEY-HERE") {
      const response = await axios.post(NEMO_GUARDRAILS_API, { text: input })
      return response.data.safe_text
    }
    return null
  }

  // Method for generating multiple words
  generateWords(input, count = 5) {
    let currentWord = input
    const words = [currentWord]

    for (let i = 1; i < count; i++) {
      const nextWord = this.generateResponse(currentWord)
      if (nextWord === "I don't know how to respond to that.") {
        break
      }
      words.push(nextWord)
      currentWord = nextWord
    }

    return words
  }
}

export default function EnhancedUnifiedHolographicNeuralNetwork() {
  const [chatHistory, setChatHistory] = useState([])
  const [inputText, setInputText] = useState('')
  const [hnn, setHnn] = useState(null)
  const [isLLMActive, setIsLLMActive] = useState(false)
  const [llmModel, setLLMModel] = useState(null)
  const [error, setError] = useState(null)
  const [isTraining, setIsTraining] = useState(false)
  const [trainingProgress, setTrainingProgress] = useState(0)
  const [learnInput, setLearnInput] = useState('')
  const [learnResponse, setLearnResponse] = useState('')
  const [isProcessing, setIsProcessing] = useState(false)
  const [progress, setProgress] = useState(0)
  const [predictedWords, setPredictedWords] = useState([])
  const [nodeId, setNodeId] = useState('')
  const [p2pConnected, setP2pConnected] = useState(false)
  const [connectedPeers, setConnectedPeers] = useState([])
  const peerRef = useRef(null)

  useEffect(() => {
    const initializeHNN = async () => {
      let storedHNN = await localforage.getItem('hnn')
      if (storedHNN) {
```

```
    const newHnn = new EnhancedHolographicNeuralNetwork(100)
    newHnn.importKnowledge(storedHNN)
    setHnn(newHnn)
  } else {
    setHnn(new EnhancedHolographicNeuralNetwork(100))
  }
}

initializeHNN()

// Initialize LLM model
const hf = new HfInference("---API---")
setLLMModel(hf)
  setIsLLMActive(true)

  // Initialize P2P connection
  initializeP2P()
}, [])

useEffect(() => {
  if (hnn) {
    const saveHNN = async () => {
      await localforage.setItem('hnn', hnn.exportKnowledge())
    }
    saveHNN()
  }
}, [hnn])

const initializeP2P = () => {
  const peer = new Peer(uuidv4())
  peerRef.current = peer

peer.on('open', (id) => {
  setNodeId(id)
  setP2pConnected(true)
})

peer.on('connection', (conn) => {
  conn.on('data', (data) => {
    handlePeerData(data)
  })
  setConnectedPeers(prev => [...prev, conn.peer])
})

peer.on('error', (error) => {
  console.error('P2P Error:', error)
  setError('P2P connection error')
  })
}

const handlePeerData = (data) => {
  if (data.type === 'knowledge') {
    hnn.importKnowledge(data.knowledge)
  }
}

const broadcastKnowledge = () => {
  const knowledge = hnn.exportKnowledge()
  connectedPeers.forEach(peerId => {
    const conn = peerRef.current.connect(peerId)
```

```
    conn.on('open', () => {
      conn.send({ type: 'knowledge', knowledge })
    })
  })
}

const handleSubmit = async (e) => {
  e.preventDefault()
  if (!inputText.trim() || !hnn) return

  let response
  let ragContext = []
  if (isLLMActive && llmModel) {
    try {
      if (useNvidiaAPIs) {
        // Use NVIDIA APIs
        const llamaIndexResult = await hnn.useLlamaIndex(inputText)
        const nemotronResult = await hnn.useNemotron70B(inputText)
        const ragResult = await hnn.useRagNvidia(inputText)
        const guardrailsResult = await hnn.useNemoGuardrails(inputText)

        // Combine results (this is a simple example, you might want to implement a more
sophisticated combination strategy)
        response = `LlamaIndex: ${llamaIndexResult}
Nemotron: ${nemotronResult}
RAG: ${ragResult}
Guardrails: ${guardrailsResult}`
      } else {
        // Use existing functionality
        ragContext = [
  { text: "Relevant context 1", score: 0.9 },
  { text: "Relevant context 2", score: 0.7 },
  { text: "Relevant context 3", score: 0.5 }
]
hnn.updateContextNodes(ragContext)

const llmResponse = await llmModel.textGeneration({
  model: 'facebook/opt-350m',
  inputs: inputText,
  parameters: {
    max_new_tokens: 50,
    temperature: 0.7,
    top_p: 0.95,
    repetition_penalty: 1.1,
  },
})
response = llmResponse.generated_text
      }
    } catch (error) {
      console.error('Error generating response:', error)
      response = "Sorry, I couldn't generate a response."
    }
  } else {
    response = hnn.generateResponse(inputText)
  }

  hnn.learn(inputText, response)
  setChatHistory(prev => [...prev, { type: 'user', text: inputText }, { type: 'bot', text:
response }])
```

```
    setInputText('')

    // Update predicted words
    const words = hnn.generateWords(inputText.split(' ').pop(), 10)
    setPredictedWords(words)

    // Broadcast updated knowledge to peers
    broadcastKnowledge()
  }

  const handleLearn = () => {
    if (learnInput.trim() && learnResponse.trim() && hnn) {
      hnn.learn(learnInput, learnResponse)
      setLearnInput('')
      setLearnResponse('')
      alert('Learning completed')
      broadcastKnowledge()
    }
  }

  const handleSave = () => {
    if (hnn) {
      const knowledge = hnn.exportKnowledge()
      const blob = new Blob([knowledge], { type: 'application/json' })
      const url = URL.createObjectURL(blob)
      const a = document.createElement('a')
      a.href = url
      a.download = 'holographic_nn_knowledge.json'
      a.click()
    }
  }

  const handleLoad = (event) => {
    const file = event.target.files?.[0]
    if (file && hnn) {
      const reader = new FileReader()
      reader.onload = (e) => {
        const knowledge = e.target?.result
        if (typeof knowledge === 'string') {
          const success = hnn.importKnowledge(knowledge)
          if (success) {
            alert('Knowledge loaded successfully')
            broadcastKnowledge()
          } else {
            alert('Error loading knowledge')
          }
        }
      }
      reader.readAsText(file)
    }
  }

  const handleTrain = () => {
    setIsTraining(true)
    setTrainingProgress(0)
    const trainStep = (i) => {
      if (i >= 100) {  // Train on 100 random samples
        setIsTraining(false)
        setTrainingProgress(0)
        alert('Training completed successfully')
```

```
    broadcastKnowledge()
    return
  }

  const input = Random input ${i}
  const output = Random output ${i}
  hnn.learn(input, output)
  setTrainingProgress(Math.round(((i + 1) / 100) * 100))

  setTimeout(() => trainStep(i + 1), 100)
}

trainStep(0)
}

const processPDF = async (file) => {
  const arrayBuffer = await file.arrayBuffer()
  const pdf = await pdfjs.getDocument(arrayBuffer).promise
  let text = ''
  for (let i = 1; i <= pdf.numPages; i++) {
    const page = await pdf.getPage(i)
    const content = await page.getTextContent()
    text += content.items.map((item) => item.str).join(' ') + ' '
    setProgress((i / pdf.numPages) * 100)
  }
  return text
}

const handleFileUpload = async (event) => {
  const file = event.target.files?.[0]
  if (file) {
    setIsProcessing(true)
    setProgress(0)
    try {
      let text = ''
      if (file.name.endsWith('.txt')) {
        text = await file.text()
      } else if (file.name.endsWith('.pdf')) {
        text = await processPDF(file)
      } else {
        throw new Error('Unsupported file type')
      }

      const words = text.toLowerCase().match(/\b\w+\b/g) || []
      for (let i = 0; i < words.length - 1; i++) {
        hnn.learn(words[i], words[i + 1])
      }

      setIsProcessing(false)
      setError(null)
      alert('File processed successfully')
      broadcastKnowledge()
    } catch (err) {
      setError('Error processing file. Please try again.')
      setIsProcessing(false)
    }
  }
}

return (
```

```
    <div className="flex h-screen">
      <div className="w-1/2 p-4 overflow-y-auto">
        <Card>
          <CardHeader>
            <CardTitle>Enhanced Holographic Neural Network Chat</CardTitle>
          </CardHeader>
          <CardContent>
            <div className="space-y-4 mb-4">
              {chatHistory.map((message, index) => (
                <div key={index} className={`flex ${message.type === 'user' ? 'justify-end' :
'justify-start'}`}>
                  <div className={`max-w-xs p-2 rounded-lg ${message.type === 'user' ?
'bg-blue-500 text-white' : 'bg-gray-200'}`}>
                    {message.text}
                  </div>
                </div>
              ))}
            </div>
            <form onSubmit={handleSubmit} className="flex space-x-2">
              <Input
                type="text"
                value={inputText}
                onChange={(e) => setInputText(e.target.value)}
                placeholder="Type your message..."
                className="flex-grow"
              />
              <Button type="submit">Send</Button>
            </form>
            {predictedWords.length > 0 && (
              <div className="mt-2 text-sm text-gray-500">
                Predicted words: {predictedWords.join(', ')}
              </div>
            )}
          </CardContent>
        </Card>
        <Card className="mt-4">
          <CardHeader>
            <CardTitle>P2P Network</CardTitle>
          </CardHeader>
          <CardContent>
            <div className="space-y-2">
              <Input
                type="text"
                value={nodeId}
                readOnly
                placeholder="Your Node ID"
              />
              <div>{p2pConnected ? 'Connected to P2P network' : 'Disconnected from P2P
network'}</div>
              <div>Connected Peers: {connectedPeers.length}</div>
            </div>
          </CardContent>
        </Card>
        <Card className="mt-4">
          <CardHeader>
            <CardTitle>Learning</CardTitle>
          </CardHeader>
          <CardContent>
            <Input
              type="text"
```

```
          value={learnInput}
          onChange={(e) => setLearnInput(e.target.value)}
          placeholder="Input to learn"
          className="mb-2"
        />
        <Input
          type="text"
          value={learnResponse}
          onChange={(e) => setLearnResponse(e.target.value)}
          placeholder="Associated response"
          className="mb-2"
        />
        <Button onClick={handleLearn}>Learn</Button>
      </CardContent>
    </Card>
    <Card className="mt-4">
      <CardHeader>
        <CardTitle>Knowledge Management</CardTitle>
      </CardHeader>
      <CardContent>
        <div className="flex space-x-2">
          <Button onClick={handleSave}>
            <Download className="mr-2 h-4 w-4" />
            Save Knowledge
          </Button>
          <Input type="file" accept=".json" onChange={handleLoad} className="hidden"
id="load-knowledge" />
          <Button onClick={() => document.getElementById('load-knowledge')?.click()}>

            <Upload className="mr-2 h-4 w-4" />
            Load Knowledge
          </Button>
        </div>
      </CardContent>
    </Card>
    <Card className="mt-4">
      <CardHeader>
        <CardTitle>Training</CardTitle>
      </CardHeader>
      <CardContent>
        <Button onClick={handleTrain} disabled={isTraining}>
          {isTraining ? 'Training...' : 'Train'}
        </Button>
        {isTraining && (
          <div className="mt-2">
            <Progress value={trainingProgress} className="w-full" />
            <p className="text-center mt-2">{trainingProgress}% completed</p>
          </div>
        )}
      </CardContent>
    </Card>
    <Card className="mt-4">
      <CardHeader>
        <CardTitle>File Processing</CardTitle>
      </CardHeader>
      <CardContent>
        <Input type="file" onChange={handleFileUpload} accept=".txt,.pdf" />
        {isProcessing && (
          <div className="mt-2">
            <Progress value={progress} className="w-full" />
```

```
          <p className="text-center mt-2">{Math.round(progress)}% processed</p>

        </div>
      )}
    </CardContent>
  </Card>
  {error && (
    <Alert variant="destructive" className="mt-4">
      <AlertCircle className="h-4 w-4" />
      <AlertTitle>Error</AlertTitle>
      <AlertDescription>{error}</AlertDescription>
    </Alert>
  )}
      </div>
      <div className="w-1/2">
        <Canvas>
          <Scene
            neurons={hnn ? hnn.neurons : []}
            connections={hnn ? hnn.connections : []}
            contextNodes={hnn ? hnn.contextNodes : []}
          />
        </Canvas>
      </div>
    </div>
  )
}
```