The Neuroscience of AI

Richard Young

2025

Chapter 18: Neuromorphic Computing

This chapter delves into neuromorphic computing systems, hardware architectures inspired by the structure and function of biological neural systems. These approaches offer potentially revolutionary advantages in energy efficiency and computational capability for specific tasks.

18.0 Chapter Goals

- Understand the principles and advantages of neuromorphic computing
- Explore spiking neural networks as a brain-inspired computational paradigm
- Learn how resistive computing elements can implement synaptic-like behavior
- Examine event-based sensing and processing paradigms
- Implement and simulate a simple spiking neural network

18.1 Neuromorphic Computing Principles

Neuromorphic computing refers to hardware systems that implement neural processing principles directly in circuitry, rather than simulating them on conventional hardware. This approach can offer dramatic improvements in energy efficiency, especially for tasks like pattern recognition and sensory processing.

Neuromorphic Computing

Key characteristics of neuromorphic systems include:

- **Parallel processing**: Massive parallelism similar to the brain's architecture
- **Co-located processing and memory**: Avoiding the von Neumann bottleneck
- **Event-driven computation**: Computing only when needed, rather than at fixed clock cycles
- **In-memory computing**: Performing operations where data is stored

- **Sparse, asynchronous signaling**: Communication through discrete events rather than continuous values

18.1.1 Spiking Neural Networks

Traditional artificial neural networks use continuous activation values, but biological neurons communicate through discrete all-or-nothing action potentials (spikes). Spiking Neural Networks (SNNs) mimic this biological principle:

```python
import numpy as np
from matplotlib import pyplot as plt

class SpikingNeuron:
    def __init__(self, threshold=1.0, tau_m=10.0, tau_ref=2.0):
        """
        Simple Leaky Integrate-and-Fire neuron model

        Parameters:
        - threshold: Membrane potential threshold for spike generation
        - tau_m: Membrane time constant (ms)
        - tau_ref: Refractory period (ms)
        """
        self.threshold = threshold
        self.tau_m = tau_m
        self.tau_ref = tau_ref

        # State variables
        self.membrane_potential = 0.0
        self.last_spike_time = -np.inf
        self.t = 0  # Current time

    def update(self, input_current, dt=1.0):
        """
        Update neuron state and check for spike

        Parameters:
        - input_current: Input current to the neuron
        - dt: Time step (ms)

        Returns:
        - 1 if neuron spikes, 0 otherwise
        """
        self.t += dt

        # Check if in refractory period
        if self.t - self.last_spike_time <= self.tau_ref:
            return 0

        # Update membrane potential (leaky integration)
        d_v = (-self.membrane_potential + input_current) / self.tau_m
        self.membrane_potential += d_v * dt

        # Check for spike
        if self.membrane_potential >= self.threshold:
            self.membrane_potential = 0.0  # Reset
            self.last_spike_time = self.t
            return 1

        return 0
```

Unlike rate-based ANNs, SNNs encode information in the precise timing of spikes and can be more energy-efficient by only computing when spikes occur. Information can be encoded in several ways:

1. **Rate coding**: Information represented by the frequency of spikes
2. **Temporal coding**: Information in the precise timing of spikes
3. **Population coding**: Information distributed across multiple neurons
4. **Rank-order coding**: Information in the order of neuron firing

18.1.2 Resistive Computing and Memristors

A key limitation in conventional computing is the energy cost of moving data between memory and processing units (the “von Neumann bottleneck”). In contrast, the brain co-locates memory and computation in synapses.

Memristors are resistive devices whose resistance changes based on the history of current flow through them. They can implement synaptic weights directly in hardware:

```python
class Memristor:
    def __init__(self, r_on=100, r_off=10000, initial_state=0.5):
        """
        Simple memristor model

        Parameters:
        - r_on: Low resistance state (ohms)
        - r_off: High resistance state (ohms)
        - initial_state: Initial state variable (0-1)
        """
        self.r_on = r_on
        self.r_off = r_off
        self.state = initial_state  # Internal state variable (0-1)

    def get_resistance(self):
        """Calculate current resistance based on internal state"""
        return self.r_on + self.state * (self.r_off - self.r_on)

    def update(self, voltage, dt=1e-6, learn_rate=1e-4):
        """
        Update memristor state based on applied voltage

        Parameters:
        - voltage: Applied voltage
        - dt: Time step
        - learn_rate: Learning rate parameter
        """
        # Simplified nonlinear update rule
        if voltage > 0:
            # Increase resistance (depression)
            self.state = min(1.0, self.state + learn_rate * voltage * dt)
        else:
            # Decrease resistance (potentiation)
            self.state = max(0.0, self.state + learn_rate * voltage * dt)
```

Memristor crossbar arrays can implement matrix multiplication operations directly in hardware with orders of magnitude less energy than digital implementations. This is especially valuable for neural network inference, where the same weights are used repeatedly.

Benefits of memristor-based computation include:

- **Energy efficiency**: 10-100× lower energy per operation
- **Density**: Higher integration density than CMOS transistors
- **Non-volatility**: Retaining state without power
- **Analog computation**: Native implementation of multiply-accumulate operations

18.1.3 Event-Based Sensors

Event-based sensors like Dynamic Vision Sensors (DVS) mimic the retina by only transmitting information when pixels detect changes in brightness:

```
def simulate_dvs_output(video_frames, threshold=0.1):
    """
    Simulate output of a Dynamic Vision Sensor from video frames

    Parameters:
    - video_frames: Sequence of image frames (T, H, W)
    - threshold: Change threshold for generating events

    Returns:
    - events: List of (x, y, t, polarity) tuples
    """
    events = []
    prev_frame = video_frames[0]

    for t, frame in enumerate(video_frames[1:], 1):
        # Calculate log intensity change
        log_diff = np.log(frame + 1e-6) - np.log(prev_frame + 1e-6)

        # Generate ON events (positive changes)
        on_events = np.where(log_diff > threshold)
        for y, x in zip(on_events[0], on_events[1]):
            events.append((x, y, t, 1))  # x, y, time, polarity

        # Generate OFF events (negative changes)
        off_events = np.where(log_diff < -threshold)
        for y, x in zip(off_events[0], off_events[1]):
            events.append((x, y, t, -1))  # x, y, time, polarity

        prev_frame = frame

    return events
```

This event-based approach drastically reduces data transmission and power requirements, enabling high-speed vision processing with minimal energy. Event-based sensors have several advantages:

- **High dynamic range**: >120dB vs. 60-70dB for conventional cameras
- **High temporal resolution**: Microsecond-level precision
- **Low bandwidth**: 10-100× less data than conventional video
- **Low latency**: Events transmitted immediately when detected
- **Low power**: 1000× more efficient than conventional imaging

18.1.4 Brain-Inspired Chips

Several neuromorphic hardware platforms have demonstrated remarkable efficiency:

1. **IBM TrueNorth**: 1 million digital neurons with 256 million synapses, consuming only ~70mW of power.
2. **Intel Loihi**: Implements on-chip learning with ~130,000 neurons and 130 million synapses per chip.
3. **SpiNNaker**: Massively parallel architecture with ARM processors designed specifically for neural simulations.
4. **BrainScaleS**: Analog/mixed-signal system operating at accelerated time scales.

These systems achieve energy efficiencies 100-1000× better than conventional architectures for certain tasks:

```
def compare_energy_efficiency():
    """
    Compare energy efficiency for image recognition task
    (based on published benchmarks)
    """
    architectures = {
        "GPU (NVIDIA V100)": {"joules_per_inference": 1.0, "accuracy": 0.76},
        "CPU (Intel Xeon)": {"joules_per_inference": 5.0, "accuracy": 0.76},
        "FPGA": {"joules_per_inference": 0.1, "accuracy": 0.75},
        "Loihi": {"joules_per_inference": 0.001, "accuracy": 0.74},
        "TrueNorth": {"joules_per_inference": 0.0001, "accuracy": 0.70}
    }

    # Calculate energy efficiency (accuracy per joule)
    for arch, stats in architectures.items():
        efficiency = stats["accuracy"] / stats["joules_per_inference"]
        print(f"{arch}: {efficiency:.1f} accuracy/joule")
```

18.2 Applications of Neuromorphic Computing

Neuromorphic systems are particularly well-suited for specific applications:

18.2.1 Edge Computing and IoT

Low-power neuromorphic chips are ideal for intelligent edge devices:

- **Sensor Processing**: Processing sensor data locally with minimal power
- **Anomaly Detection**: Identifying unusual patterns without continuous transmission
- **Keyword Spotting**: Recognizing specific audio triggers
- **Smart Cameras**: Event-based vision for surveillance and monitoring

18.2.2 Robotics and Autonomous Systems

Neuromorphic computing enables efficient sensorimotor processing:

- **Real-time Control**: Low-latency sensory processing and actuation
- **Obstacle Avoidance**: Fast processing of visual information for navigation
- **Power Efficiency**: Extended operation time on limited power budgets
- **Adaptive Behavior**: On-chip learning for environmental adaptation

18.2.3 Brain-Computer Interfaces

The low power and event-driven nature of neuromorphic systems makes them ideal for neural interfaces:

- **Neural Signal Processing**: Efficient processing of sparse neural signals
- **Closed-loop Stimulation**: Real-time response to detected neural patterns
- **Portable Medical Devices**: Neurological monitoring with long battery life
- **Neuroprosthetics**: Direct processing of neural signals for prosthetic control

18.3 Simulation and Implementation

18.3.1 Software Frameworks for Neuromorphic Computing

Several frameworks support the development and simulation of spiking neural networks:

1. **Brian2**: Python-based simulator for spiking neural networks
2. **NEST**: Simulator for large-scale networks of spiking neurons

3. **PyNN**: API for simulator-independent specification of neural network models
4. **Nengo**: Python library for building and simulating neural models
5. **BindsNET**: SNN framework built on PyTorch
6. **Norse**: Deep learning with spiking neural networks in PyTorch

18.3.2 Converting ANNs to SNNs

Converting traditional artificial neural networks to spiking neural networks allows leveraging existing deep learning methods:

```
def convert_ann_to_snn(ann_model, simulation_time=100, dt=1.0):
    """
    Convert a trained ANN to a rate-based SNN

    Parameters:
    - ann_model: Trained artificial neural network
    - simulation_time: Simulation duration in ms
    - dt: Time step in ms

    Returns:
    - snn_model: Equivalent spiking neural network
    """
    # In practice, this would involve:
    # 1. Extracting weights from ANN
    # 2. Creating appropriate SNN architecture
    # 3. Setting thresholds based on activation statistics
    # 4. Adjusting for rate-based operation

    # Placeholder implementation
    snn_model = create_empty_snn_model()

    # For each layer in the ANN
    for layer_idx, layer in enumerate(ann_model.layers):
        if hasattr(layer, 'weight'):
            # Copy weights
            weights = layer.weight.data.numpy()
            set_snn_weights(snn_model, layer_idx, weights)

            # Set appropriate thresholds based on activation statistics
            activation_scale = estimate_activation_scale(ann_model, layer_idx)
            set_snn_thresholds(snn_model, layer_idx, activation_scale)

    return snn_model
```

Key challenges in ANN-to-SNN conversion include:

- **Activation function mapping**: Converting ReLU to spike rates
- **Threshold calibration**: Setting appropriate firing thresholds
- **Temporal dynamics**: Handling the time dimension
- **Training-aware conversion**: Optimizing ANNs specifically for SNN conversion

18.4 Code Lab: Spiking Neural Network

Let's implement a simple spiking neural network that demonstrates the key principles of neuromorphic computing:

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

class SpikingNeuralNetwork:
    def __init__(self, n_input, n_hidden, n_output, dt=1.0):
        """
        Simple Spiking Neural Network with LIF neurons

        Parameters:
        - n_input: Number of input neurons
        - n_hidden: Number of hidden neurons
        - n_output: Number of output neurons
        - dt: Simulation time step (ms)
        """
        self.n_input = n_input
        self.n_hidden = n_hidden
        self.n_output = n_output
        self.dt = dt

        # Initialize random weights
        self.w_input_hidden = np.random.normal(0, 0.1, (n_hidden, n_input))
        self.w_hidden_output = np.random.normal(0, 0.1, (n_output, n_hidden))

        # Neuron parameters
        self.v_rest = -70.0  # resting potential (mV)
        self.v_reset = -75.0  # reset potential (mV)
        self.v_threshold = -55.0  # threshold potential (mV)
        self.tau_m = 10.0  # membrane time constant (ms)
        self.refractory_period = 2.0  # refractory period (ms)

        # State variables
        self.v_hidden = np.ones(n_hidden) * self.v_rest
        self.v_output = np.ones(n_output) * self.v_rest
        self.refractory_time_hidden = np.zeros(n_hidden)
        self.refractory_time_output = np.zeros(n_output)

        # Recording
        self.spike_history_hidden = []
        self.spike_history_output = []
        self.v_history_hidden = []
        self.v_history_output = []

        self.t = 0  # Current time

    def reset_state(self):
        """Reset network state"""
        self.v_hidden = np.ones(self.n_hidden) * self.v_rest
        self.v_output = np.ones(self.n_output) * self.v_rest
        self.refractory_time_hidden = np.zeros(self.n_hidden)
        self.refractory_time_output = np.zeros(self.n_output)
        self.spike_history_hidden = []
        self.spike_history_output = []
```

```
        self.v_history_hidden = []
        self.v_history_output = []
        self.t = 0

    def step(self, input_spikes):
        """
        Run one simulation step

        Parameters:
        - input_spikes: Binary array of input spikes (0 or 1)

        Returns:
        - output_spikes: Binary array of output spikes
        """
        self.t += self.dt

        # Update hidden layer
        # Calculate input current from input layer spikes
        input_current = np.dot(self.w_input_hidden, input_spikes)

        # Check which neurons are not in refractory period
        active_hidden = self.refractory_time_hidden <= 0

        # Update membrane potentials for active neurons
        dv_hidden = (-self.v_hidden + self.v_rest + input_current) / self.tau_m
        self.v_hidden[active_hidden] += dv_hidden[active_hidden] * self.dt

        # Check for spikes in hidden layer
        hidden_spikes = (self.v_hidden >= self.v_threshold).astype(int)

        # Reset membrane potential and set refractory period for spiked neurons
        if np.any(hidden_spikes):
            spike_indices = np.where(hidden_spikes)[0]
            self.v_hidden[spike_indices] = self.v_reset
            self.refractory_time_hidden[spike_indices] = self.refractory_period

        # Decrement refractory time
        self.refractory_time_hidden -= self.dt
        self.refractory_time_hidden = np.maximum(0, self.refractory_time_hidden)

        # Update output layer (similar process)
        output_current = np.dot(self.w_hidden_output, hidden_spikes)
        active_output = self.refractory_time_output <= 0

        dv_output = (-self.v_output + self.v_rest + output_current) / self.tau_m
        self.v_output[active_output] += dv_output[active_output] * self.dt

        output_spikes = (self.v_output >= self.v_threshold).astype(int)

        if np.any(output_spikes):
            spike_indices = np.where(output_spikes)[0]
            self.v_output[spike_indices] = self.v_reset
            self.refractory_time_output[spike_indices] = self.refractory_period
```

```
        self.refractory_time_output -= self.dt
        self.refractory_time_output = np.maximum(0, self.refractory_time_output)

        # Record history
        self.spike_history_hidden.append(hidden_spikes.copy())
        self.spike_history_output.append(output_spikes.copy())
        self.v_history_hidden.append(self.v_hidden.copy())
        self.v_history_output.append(self.v_output.copy())

        return output_spikes

    def run_simulation(self, input_pattern, simulation_time=100):
        """
        Run simulation for specified time with given input pattern

        Parameters:
        - input_pattern: Function that returns input spikes at each time step
        - simulation_time: Total simulation time (ms)

        Returns:
        - output_history: History of output spikes
        """
        steps = int(simulation_time / self.dt)
        self.reset_state()

        output_history = []

        for step in range(steps):
            t = step * self.dt
            input_spikes = input_pattern(t)
            output_spikes = self.step(input_spikes)
            output_history.append(output_spikes)

        return np.array(output_history)

    def plot_activity(self, figsize=(12, 8)):
        """Plot spike raster and membrane potentials"""
        if not self.spike_history_hidden:
            print("No simulation data to plot")
            return

        fig, axs = plt.subplots(4, 1, figsize=figsize, sharex=True)

        # Convert spike history to arrays
        spikes_hidden = np.array(self.spike_history_hidden)
        spikes_output = np.array(self.spike_history_output)

        # Time array
        time = np.arange(0, len(spikes_hidden) * self.dt, self.dt)

        # Plot hidden layer spikes
        for i in range(self.n_hidden):
            spike_times = time[spikes_hidden[:, i] > 0]
            axs[0].scatter(spike_times, np.ones_like(spike_times) * i, color='bla
```

```
        axs[0].set_ylabel('Hidden Neuron')
        axs[0].set_title('Hidden Layer Spike Raster')

        # Plot output layer spikes
        for i in range(self.n_output):
            spike_times = time[spikes_output[:, i] > 0]
            axs[1].scatter(spike_times, np.ones_like(spike_times) * i, color='red
        axs[1].set_ylabel('Output Neuron')
        axs[1].set_title('Output Layer Spike Raster')

        # Plot membrane potentials
        v_hidden = np.array(self.v_history_hidden)
        v_output = np.array(self.v_history_output)

        # Plot a few hidden neurons
        for i in range(min(3, self.n_hidden)):
            axs[2].plot(time, v_hidden[:, i], label=f'Neuron {i}')
        axs[2].axhline(y=self.v_threshold, color='r', linestyle='--', label='Thre
        axs[2].set_ylabel('Membrane Potential (mV)')
        axs[2].set_title('Hidden Layer Membrane Potentials')
        axs[2].legend()

        # Plot all output neurons
        for i in range(self.n_output):
            axs[3].plot(time, v_output[:, i], label=f'Neuron {i}')
        axs[3].axhline(y=self.v_threshold, color='r', linestyle='--', label='Thre
        axs[3].set_xlabel('Time (ms)')
        axs[3].set_ylabel('Membrane Potential (mV)')
        axs[3].set_title('Output Layer Membrane Potentials')
        axs[3].legend()

        plt.tight_layout()
        return fig
```

Let’s demonstrate this SNN with a simple pattern recognition task:

```python
def run_snn_demo():
    # Create a simple SNN with 5 input, 10 hidden, and 2 output neurons
    snn = SpikingNeuralNetwork(5, 10, 2)

    # Define input patterns (simplified)
    def pattern_1(t):
        # Pattern 1: neurons 0, 1, 2 active
        period = 20  # ms
        return np.array([1 if t % period < 5 else 0,
                        1 if (t % period) > 3 and (t % period) < 8 else 0,
                        1 if (t % period) > 6 and (t % period) < 12 else 0,
                        0, 0])

    def pattern_2(t):
        # Pattern 2: neurons 2, 3, 4 active
        period = 20  # ms
        return np.array([0, 0,
                        1 if (t % period) > 2 and (t % period) < 7 else 0,
                        1 if (t % period) > 5 and (t % period) < 10 else 0,
                        1 if (t % period) > 8 and (t % period) < 15 else 0])

    # Run simulations with different input patterns
    print("Running simulation for pattern 1...")
    snn.run_simulation(pattern_1, 200)
    fig1 = snn.plot_activity()
    plt.figure(fig1.number)
    plt.suptitle("Pattern 1 Response")

    print("Running simulation for pattern 2...")
    snn.run_simulation(pattern_2, 200)
    fig2 = snn.plot_activity()
    plt.figure(fig2.number)
    plt.suptitle("Pattern 2 Response")

    plt.show()

    return "SNN demo completed"
```

18.5 Future Directions

Neuromorphic computing is evolving rapidly with several exciting research directions:

18.5.1 Hardware Advances

- **3D Integration**: Increasing neuron and synapse density through vertical stacking

- **Novel Materials**: Exploring phase-change memory, ferroelectric devices, and spintronic devices
- **Hybrid Systems**: Combining neuromorphic and conventional computing architectures
- **Nanoscale Devices**: Moving toward truly brain-like densities with nanoscale components

18.5.2 Algorithms and Learning

- **Local Learning Rules**: Developing biologically plausible learning algorithms that don't require global backpropagation
- **Temporal Coding**: Moving beyond rate coding to leverage timing information
- **Structural Plasticity**: Algorithms that modify network topology, not just weights
- **Neuromodulatory Influences**: Incorporating attention, reward, and other modulatory effects

18.5.3 System Integration

- **Sensorimotor Loops**: Closing the loop between sensing and acting in neuromorphic systems
- **Hybrid Learning**: Combining traditional deep learning with neuromorphic approaches
- **Multi-chip Systems**: Scaling to brain-like numbers of neurons and synapses
- **Standardized Interfaces**: Developing common interfaces for neuromorphic hardware

18.6 Take-aways

- **Neuromorphic computing mimics the brain's architecture** to achieve dramatic improvements in energy efficiency for specific types of computation.
- **Spiking neural networks** use discrete, event-based communication similar to biological neurons, enabling sparse computation.
- **Resistive computing elements** like memristors can implement synaptic weights directly in hardware, overcoming the von Neumann bottleneck.
- **Event-based sensors** drastically reduce data bandwidth and power requirements by only transmitting information when changes occur.
- **Neuromorphic hardware platforms** demonstrate 100-1000× improvements in energy efficiency for pattern recognition and sensory processing tasks.

18.7 Further Reading

- Davies, M., et al. (2018). Loihi: A Neuromorphic Manycore Processor with On-Chip Learning. IEEE Micro, 38(1), 82-99.
- Schuman, C. D., et al. (2017). A Survey of Neuromorphic Computing and Neural Networks in Hardware. arXiv preprint arXiv:1705.06963.
- Diehl, P. U., et al. (2015). Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing. IEEE International Joint Conference on Neural Networks.
- Tavanaei, A., et al. (2019). Deep learning in spiking neural networks. Neural Networks, 111, 47-63.
- Pfeiffer, M., & Pfeil, T. (2018). Deep learning with spiking neurons: Opportunities and challenges. Frontiers in Neuroscience, 12, 774.