Chapter 21: AI for Neuroscience Discovery

This chapter examines how artificial intelligence is accelerating neuroscience research and enabling new discoveries about brain function and organization. While neuroscience has heavily inspired AI, AI is now increasingly being used to advance neuroscience in a virtuous cycle of innovation. This includes not only fundamental research but also clinical applications that translate neuroscience insights into healthcare solutions for neurological disorders and conditions.

21.0 Chapter Goals

- Understand how AI approaches are transforming neural data analysis
- Explore computational brain simulations enabled by machine learning
- Learn how AI assists in connectome reconstruction and cellular morphology analysis
- Discover how machine learning helps develop theories of brain function
- Implement neural data analysis tools using deep learning techniques
- Apply AI techniques to clinical neuroimaging for disease diagnosis and treatment planning

21.1 Neural Data Analysis with Deep Learning

The exponential growth in neural recording technologies has created a data analysis challenge that AI is uniquely positioned to address. Deep learning approaches can extract patterns and insights from complex, high-dimensional neural data that would be difficult or impossible to identify with traditional methods.

21.1.0 Clinical Neuroimaging Analysis

Modern healthcare relies heavily on neuroimaging for diagnosis, treatment planning, and monitoring of neurological conditions. AI techniques have revolutionized these analyses, making them more accurate, efficient, and clinically applicable.

```python
def analyze_clinical_neuroimaging(imaging_data, modality="fMRI", task="classificatic
    """
    Analyze clinical neuroimaging data with deep learning approaches

    Parameters:
    - imaging_data: Neuroimaging data (format depends on modality)
    - modality: Imaging modality ('fMRI', 'EEG', 'MRI', 'CT', 'PET')
    - task: Analysis task ('classification', 'segmentation', 'prediction')

    Returns:
    - results: Analysis results and visualization
    """
    import numpy as np
    import tensorflow as tf
    from tensorflow.keras import layers, Model
    import matplotlib.pyplot as plt

    # Preprocessing based on modality
    if modality == "fMRI":
        # Spatial and temporal preprocessing for fMRI
        preprocessed_data = preprocess_fmri(imaging_data)
        model = build_fmri_model(task)
    elif modality == "EEG":
        # Filter and artifact removal for EEG
        preprocessed_data = preprocess_eeg(imaging_data)
        model = build_eeg_model(task)
    elif modality in ["MRI", "CT", "PET"]:
        # Structural image preprocessing
        preprocessed_data = preprocess_structural(imaging_data, modality)
        model = build_structural_model(modality, task)
    else:
        raise ValueError(f"Unsupported modality: {modality}")

    # Perform analysis based on task
    if task == "classification":
        # Disease classification (e.g., Alzheimer's, tumor, stroke)
        results = perform_classification(model, preprocessed_data)
    elif task == "segmentation":
        # Segment abnormalities (e.g., lesions, tumors)
        results = perform_segmentation(model, preprocessed_data)
    elif task == "prediction":
        # Predict disease progression or treatment response
        results = perform_prediction(model, preprocessed_data)
    else:
        raise ValueError(f"Unsupported task: {task}")

    return results

def build_fmri_model(task):
    """Build a deep learning model for fMRI analysis"""
    inputs = tf.keras.Input(shape=(91, 109, 91, 20))  # Example: (x, y, z, time)

    # 3D CNN + LSTM for spatiotemporal features
    x = layers.Conv3D(32, kernel_size=3, activation="relu")(inputs)
    x = layers.MaxPooling3D(pool_size=2)(x)
    x = layers.Conv3D(64, kernel_size=3, activation="relu")(x)
    x = layers.MaxPooling3D(pool_size=2)(x)

    # Reshape for sequence processing
    x = layers.Reshape((-1, 64))(x)
    x = layers.LSTM(128, return_sequences=False)(x)

    if task == "classification":
        outputs = layers.Dense(2, activation="softmax")(x)  # Binary classification
    elif task == "segmentation":
        # Upsampling path for segmentation
        x = layers.Dense(91*109*91, activation="relu")(x)
        x = layers.Reshape((91, 109, 91))(x)
        outputs = layers.Conv3D(1, kernel_size=1, activation="sigmoid")(x)
    else:  # prediction
        outputs = layers.Dense(1, activation="linear")(x)  # Regression

    return Model(inputs=inputs, outputs=outputs)

def build_eeg_model(task):
    """Build a deep learning model for EEG analysis"""
    inputs = tf.keras.Input(shape=(64, 1000))  # Example: (channels, time)

    # Temporal convolutional network
    x = layers.Conv1D(32, kernel_size=10, activation="relu")(inputs)
    x = layers.MaxPooling1D(pool_size=2)(x)
    x = layers.Conv1D(64, kernel_size=10, activation="relu")(x)
    x = layers.MaxPooling1D(pool_size=2)(x)
    x = layers.Conv1D(128, kernel_size=10, activation="relu")(x)
    x = layers.GlobalAveragePooling1D()(x)

    if task == "classification":
        outputs = layers.Dense(2, activation="softmax")(x)  # Binary classification
    else:  # prediction
        outputs = layers.Dense(1, activation="linear")(x)  # Regression

    return Model(inputs=inputs, outputs=outputs)

def alzheimers_fmri_classification():
    """Example of Alzheimer's disease classification from fMRI data"""
    import numpy as np

    # Simulate data for demonstration (in practice, would load real data)
    n_samples = 100
    n_x, n_y, n_z, n_t = 91, 109, 91, 20

    # Generate synthetic data
    X = np.random.randn(n_samples, n_x, n_y, n_z, n_t) * 0.1

    # Add disease-specific patterns to half the samples
    y = np.zeros(n_samples)
    for i in range(n_samples // 2, n_samples):
        # Simulate hippocampal atrophy in Alzheimer's patients
        pattern = np.zeros((n_x, n_y, n_z, n_t))
        pattern[40:50, 45:55, 40:50, :] = np.random.randn(10, 10, 10, n_t) * 0.5
        X[i] += pattern
        y[i] = 1  # Alzheimer's positive

    # Build model
    model = build_fmri_model("classification")
    model.compile(optimizer="adam", loss="sparse_categorical_crossentropy", metrics=

    # Train/test split
    from sklearn.model_selection import train_test_split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_

    # Train model (in practice, would use more epochs)
    model.fit(X_train, y_train, batch_size=8, epochs=5, validation_split=0.2)

    # Evaluate
    results = model.evaluate(X_test, y_test)
    print(f"Test accuracy: {results[1]:.4f}")

    # Generate visualization for model interpretation
    grad_cam_visualization(model, X_test[0:1], class_idx=1)

    return model, results

def epilepsy_eeg_detection():
    """Example of epileptic seizure detection from EEG data"""
    import numpy as np

    # Simulate data for demonstration
    n_samples = 200
    n_channels, n_timepoints = 64, 1000

    # Generate synthetic data
    X = np.random.randn(n_samples, n_channels, n_timepoints) * 0.1

    # Add seizure patterns to half the samples
    y = np.zeros(n_samples)
    for i in range(n_samples // 2, n_samples):
        # Simulate spike-and-wave discharges characteristic of seizures
        for j in range(5, n_timepoints, 20):
            # Add spike pattern
            X[i, :, j:j+3] += np.random.randn(n_channels, 3) * 2.0
            # Add slow wave
            t = np.arange(10) / 10.0
            wave = np.sin(2 * np.pi * t)
            X[i, :, j+3:j+13] += np.expand_dims(wave, 0) * 0.5
        y[i] = 1  # Seizure positive

    # Build model
    model = build_eeg_model("classification")
    model.compile(optimizer="adam", loss="sparse_categorical_crossentropy", metrics=
    # Train/test split
    from sklearn.model_selection import train_test_split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_

    # Train model (in practice, would use more epochs)
    model.fit(X_train, y_train, batch_size=16, epochs=5, validation_split=0.2)

    # Evaluate
    results = model.evaluate(X_test, y_test)
    print(f"Test accuracy: {results[1]:.4f}")

    # Visualize model attention
    visualize_eeg_attention(model, X_test[0], y_test[0])

    return model, results

def stroke_outcome_prediction():
    """Example of stroke outcome prediction from multimodal imaging"""
    import numpy as np

    # Simulate data for demonstration
    n_samples = 150
    n_x, n_y, n_z = 91, 109, 91

    # Generate synthetic structural MRI data
    X_mri = np.random.randn(n_samples, n_x, n_y, n_z) * 0.1

    # Generate synthetic diffusion tensor imaging (DTI) data
    X_dti = np.random.randn(n_samples, n_x, n_y, n_z, 6) * 0.1  # 6 DTI measures

    # Add lesion patterns with varying sizes
    lesion_sizes = np.random.randint(5, 20, size=n_samples)

    # Create outcome scores (modified Rankin Scale, 0-6 where higher is worse)
    y = np.zeros(n_samples)

    for i in range(n_samples):
        lesion_size = lesion_sizes[i]

        # Random lesion location
        x_loc = np.random.randint(30, 60)
        y_loc = np.random.randint(30, 60)
        z_loc = np.random.randint(30, 60)

        # Add lesion to MRI
        X_mri[i, x_loc:x_loc+lesion_size, y_loc:y_loc+lesion_size, z_loc:z_loc+lesio

        # Add corresponding DTI changes
        X_dti[i, x_loc:x_loc+lesion_size, y_loc:y_loc+lesion_size, z_loc:z_loc+lesio

        # Outcome depends on lesion size and location
        # Lesions near center (motor pathways) are more impactful
        center_distance = np.sqrt((x_loc-45)**2 + (y_loc-45)**2 + (z_loc-45)**2)
        location_factor = 1.0 - (center_distance / 50.0)  # Normalize to 0-1
        # Compute outcome score
        outcome = (lesion_size / 20.0) * 6 * location_factor
        y[i] = min(6, max(0, outcome + np.random.randn() * 0.5))  # Add noise

    # Build a multimodal model
    inputs_mri = tf.keras.Input(shape=(n_x, n_y, n_z))
    inputs_dti = tf.keras.Input(shape=(n_x, n_y, n_z, 6))

    # MRI processing branch
    x_mri = layers.Conv3D(16, kernel_size=3, activation="relu")(inputs_mri[:,:,:,:,t
    x_mri = layers.MaxPooling3D(pool_size=2)(x_mri)
    x_mri = layers.Conv3D(32, kernel_size=3, activation="relu")(x_mri)
    x_mri = layers.GlobalAveragePooling3D()(x_mri)

    # DTI processing branch
    x_dti = layers.Conv3D(16, kernel_size=3, activation="relu")(inputs_dti)
    x_dti = layers.MaxPooling3D(pool_size=2)(x_dti)
    x_dti = layers.Conv3D(32, kernel_size=3, activation="relu")(x_dti)
    x_dti = layers.GlobalAveragePooling3D()(x_dti)

    # Combine modalities
    x = layers.concatenate([x_mri, x_dti])
    x = layers.Dense(64, activation="relu")(x)
    outputs = layers.Dense(1, activation="linear")(x)  # Regression

    model = Model(inputs=[inputs_mri, inputs_dti], outputs=outputs)
    model.compile(optimizer="adam", loss="mse", metrics=["mae"])

    # Train/test split
    from sklearn.model_selection import train_test_split
    X_train_mri, X_test_mri, X_train_dti, X_test_dti, y_train, y_test = train_test_s
        X_mri, X_dti, y, test_size=0.3, random_state=42)

    # Train model
    model.fit(
        [X_train_mri, X_train_dti], y_train,
        batch_size=8, epochs=5, validation_split=0.2
    )

    # Evaluate
    results = model.evaluate([X_test_mri, X_test_dti], y_test)
    print(f"Test MAE: {results[1]:.4f}")

    # Visualize feature importance
    visualize_feature_importance(model, [X_test_mri[0:1], X_test_dti[0:1]])

    return model, results
```

Clinical applications of AI for neuroimaging analysis include:

1. **Disease Classification**:

    - Alzheimer’s disease and dementia detection
    - Brain tumor classification
    - Stroke diagnosis
    - Epilepsy focus localization
2. **Disease Progression Monitoring**:
    - Multiple sclerosis lesion tracking
    - Parkinson’s disease progression
    - Post-stroke recovery assessment
    - Treatment response prediction
3. **Anatomical Segmentation**:
    - Tumor boundary delineation
    - White matter lesion quantification
    - Hippocampal volumetry for dementia
    - Cortical thickness measurement
4. **Multimodal Integration**:
    - Combining structural MRI with functional data
    - Integrating EEG with fMRI for epilepsy
    - Fusing PET and MRI for Alzheimer’s characterization
    - Multiparametric analysis for enhanced diagnosis

21.1.1 Decoding Neural Activity

Deep learning models can decode neural activity to predict behavior, perception, or cognitive states:

```python
def analyze_neural_recordings(spike_data, behavior_data):
    """
    Use deep learning to analyze neural recording data

    Parameters:
    - spike_data: Neural spike recordings [neurons, time]
    - behavior_data: Behavioral measurements [time, features]

    Returns:
    - model: Trained neural decoder
    """
    import torch
    import torch.nn as nn
    import torch.optim as optim
    import numpy as np

    # Prepare data
    X = spike_data.T  # [time, neurons]
    y = behavior_data  # [time, features]

    # Split into train/test
    split_idx = int(0.8 * len(X))
    X_train, X_test = X[:split_idx], X[split_idx:]
    y_train, y_test = y[:split_idx], y[split_idx:]

    # Create and train a neural decoder
    model = nn.Sequential(
        nn.Linear(X.shape[1], 128),
        nn.ReLU(),
        nn.Dropout(0.5),
        nn.Linear(128, 64),
        nn.ReLU(),
        nn.Linear(64, y.shape[1])
    )

    # Train the model
    optimizer = optim.Adam(model.parameters(), lr=0.001)
    criterion = nn.MSELoss()

    for epoch in range(100):
        optimizer.zero_grad()
        outputs = model(torch.tensor(X_train, dtype=torch.float32))
        loss = criterion(outputs, torch.tensor(y_train, dtype=torch.float32))
        loss.backward()
        optimizer.step()

    # Evaluate
    with torch.no_grad():
        y_pred = model(torch.tensor(X_test, dtype=torch.float32)).numpy()
        r2_scores = [np.corrcoef(y_test[:, i], y_pred[:, i])[0, 1]**2
                    for i in range(y_test.shape[1])]
        print(f"Average R² score: {np.mean(r2_scores):.3f}")
    return model
```

These decoding models can reveal how neural populations represent information and how these representations evolve over time. Applications include:

- **Movement Decoding**: Predicting limb trajectories from motor cortex activity
- **Sensory Reconstruction**: Reconstructing visual or auditory stimuli from brain activity
- **Cognitive State Classification**: Identifying decision-making processes, attention states, or memory formation

21.1.2 Dimensionality Reduction for Neural Data

Neural data is often high-dimensional, with recordings from hundreds or thousands of neurons. Deep learning approaches can identify lower-dimensional representations that capture the essential dynamics:

```python
def neural_dimensionality_reduction(neural_data, latent_dim=10):
    """
    Reduce dimensionality of neural data using a variational autoencoder

    Parameters:
    - neural_data: Neural activity data [samples, neurons]
    - latent_dim: Dimension of latent space

    Returns:
    - encoder: Model that maps from neural activity to latent space
    - decoder: Model that maps from latent space to neural activity
    - latent_representations: Neural data in latent space
    """
    import torch
    import torch.nn as nn
    import torch.optim as optim
    from torch.nn import functional as F

    # Normalize data
    data_mean = neural_data.mean(axis=0, keepdims=True)
    data_std = neural_data.std(axis=0, keepdims=True) + 1e-6
    normalized_data = (neural_data - data_mean) / data_std

    # Convert to torch tensor
    data_tensor = torch.tensor(normalized_data, dtype=torch.float32)

    # Define VAE architecture
    class VAE(nn.Module):
        def __init__(self, input_dim, latent_dim):
            super(VAE, self).__init__()

            # Encoder
            self.fc1 = nn.Linear(input_dim, 128)
            self.fc2 = nn.Linear(128, 64)
            self.fc_mu = nn.Linear(64, latent_dim)
            self.fc_logvar = nn.Linear(64, latent_dim)

            # Decoder
            self.fc3 = nn.Linear(latent_dim, 64)
            self.fc4 = nn.Linear(64, 128)
            self.fc5 = nn.Linear(128, input_dim)

        def encode(self, x):
            h = F.relu(self.fc1(x))
            h = F.relu(self.fc2(h))
            return self.fc_mu(h), self.fc_logvar(h)

        def reparameterize(self, mu, logvar):
            std = torch.exp(0.5 * logvar)
            eps = torch.randn_like(std)
            return mu + eps * std

        def decode(self, z):
            h = F.relu(self.fc3(z))
            h = F.relu(self.fc4(h))
            return self.fc5(h)

        def forward(self, x):
            mu, logvar = self.encode(x)
            z = self.reparameterize(mu, logvar)
            return self.decode(z), mu, logvar

    # Initialize model
    input_dim = neural_data.shape[1]
    vae = VAE(input_dim, latent_dim)

    # Train VAE
    optimizer = optim.Adam(vae.parameters(), lr=0.001)

    # VAE loss function
    def loss_function(recon_x, x, mu, logvar):
        BCE = F.mse_loss(recon_x, x, reduction='sum')
        KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
        return BCE + KLD

    # Training loop
    for epoch in range(100):
        optimizer.zero_grad()
        recon_batch, mu, logvar = vae(data_tensor)
        loss = loss_function(recon_batch, data_tensor, mu, logvar)
        loss.backward()
        optimizer.step()

        if epoch % 10 == 0:
            print(f'Epoch {epoch}, Loss: {loss.item() / len(data_tensor):.4f}')

    # Extract latent representations
    with torch.no_grad():
        mu, _ = vae.encode(data_tensor)
        latent_representations = mu.numpy()

    # Create encoder and decoder functions
    def encoder(data):
        data_norm = (data - data_mean) / data_std
        with torch.no_grad():
            mu, _ = vae.encode(torch.tensor(data_norm, dtype=torch.float32))
        return mu.numpy()

    def decoder(latent):
        with torch.no_grad():
            recon = vae.decode(torch.tensor(latent, dtype=torch.float32))
        recon_unnorm = recon.numpy() * data_std + data_mean
        return recon_unnorm

    return encoder, decoder, latent_representations
```

Dimensionality reduction techniques like variational autoencoders (VAEs) can:

- Identify low-dimensional neural manifolds that represent behavioral states
- Reveal population-level dynamics not visible at the single-neuron level
- Enable visualization of high-dimensional neural trajectories
- Discover shared structure across different brain regions or subjects

21.1.3 Time Series Analysis for Neural Dynamics

Deep learning models are particularly effective for analyzing the temporal dynamics of neural activity:

```python
def analyze_neural_dynamics(neural_time_series, sequence_length=50):
    """
    Analyze neural dynamics using a recurrent neural network

    Parameters:
    - neural_time_series: Neural activity over time [time, neurons]
    - sequence_length: Length of sequences for prediction

    Returns:
    - model: Trained RNN model for neural dynamics prediction
    """
    import torch
    import torch.nn as nn
    import torch.optim as optim
    import numpy as np

    # Prepare sequence data
    def create_sequences(data, seq_length):
        X, y = [], []
        for i in range(len(data) - seq_length):
            X.append(data[i:i+seq_length])
            y.append(data[i+seq_length])
        return np.array(X), np.array(y)

    X, y = create_sequences(neural_time_series, sequence_length)

    # Split into train/test
    split_idx = int(0.8 * len(X))
    X_train, X_test = X[:split_idx], X[split_idx:]
    y_train, y_test = y[:split_idx], y[split_idx:]

    # Define RNN model
    class NeuralRNN(nn.Module):
        def __init__(self, input_size, hidden_size, output_size):
            super(NeuralRNN, self).__init__()
            self.hidden_size = hidden_size
            self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
            self.fc = nn.Linear(hidden_size, output_size)

        def forward(self, x):
            lstm_out, _ = self.lstm(x)
            return self.fc(lstm_out[:, -1, :])

    # Model parameters
    input_size = neural_time_series.shape[1]  # Number of neurons
    hidden_size = 64
    output_size = input_size

    # Initialize model
    model = NeuralRNN(input_size, hidden_size, output_size)

    # Train model
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    # Convert to torch tensors
    X_train_t = torch.tensor(X_train, dtype=torch.float32)
    y_train_t = torch.tensor(y_train, dtype=torch.float32)
    X_test_t = torch.tensor(X_test, dtype=torch.float32)
    y_test_t = torch.tensor(y_test, dtype=torch.float32)

    # Training loop
    for epoch in range(100):
        # Forward pass
        y_pred = model(X_train_t)
        loss = criterion(y_pred, y_train_t)

        # Backward pass and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if epoch % 10 == 0:
            # Evaluate on test set
            with torch.no_grad():
                test_pred = model(X_test_t)
                test_loss = criterion(test_pred, y_test_t).item()
            print(f'Epoch {epoch}, Train Loss: {loss.item():.4f}, Test Loss: {test_l

    # Evaluate prediction performance
    with torch.no_grad():
        y_pred = model(X_test_t).numpy()
        r2_scores = [np.corrcoef(y_test[:, i], y_pred[:, i])[0, 1]**2
                     for i in range(y_test.shape[1])]
        print(f"Average R² score: {np.mean(r2_scores):.3f}")

    return model
```

RNN-based models can:

- Predict future neural activity based on past patterns
- Identify recurrent dynamics and attractor states
- Characterize how external inputs perturb ongoing neural dynamics
- Model the temporal evolution of neural representations

21.2 Brain Simulation Efforts

AI is enabling increasingly detailed and realistic simulations of brain activity, from single neurons to large-scale networks.

21.2.1 Large-Scale Neural Circuit Simulations

AI-assisted models can simulate the activity of large populations of neurons:

```
class BrainRegionSimulation:
    def __init__(self, n_neurons=1000, connectivity_density=0.1):
        """
        Simplified brain region simulation

        Parameters:
        - n_neurons: Number of neurons to simulate
        - connectivity_density: Fraction of possible connections to create
        """
        import numpy as np

        self.n_neurons = n_neurons

        # Initialize neurons (simplified LIF model)
        self.v_rest = -70.0  # resting potential (mV)
        self.v_threshold = -55.0  # spike threshold (mV)
        self.v_reset = -75.0  # reset potential (mV)
        self.tau = 20.0  # membrane time constant (ms)

        # State variables
        self.v = np.ones(n_neurons) * self.v_rest  # membrane potentials
        self.refractory_time = np.zeros(n_neurons)  # time until end of refractory p

        # Generate random connectivity matrix
        p = connectivity_density
        self.weights = np.random.choice(
            [0, 1], size=(n_neurons, n_neurons), p=[1-p, p]
        )
        # Scale weights and ensure no self-connections
        self.weights = self.weights * np.random.normal(0, 0.1, (n_neurons, n_neurons
        np.fill_diagonal(self.weights, 0)

        # 80% excitatory, 20% inhibitory
        inh_neurons = np.random.choice(n_neurons, size=int(0.2 * n_neurons), replace
        self.weights[inh_neurons] *= -5

        # Record spikes
        self.spike_times = [[] for _ in range(n_neurons)]
        self.current_time = 0

    def step(self, external_input=None, dt=0.1):
        """
        Simulate one time step

        Parameters:
        - external_input: External current to each neuron
        - dt: Time step (ms)

        Returns:
        - spikes: Boolean array indicating which neurons spiked
        """
        import numpy as np
```

```python
        self.current_time += dt

        # Default to no external input
        if external_input is None:
            external_input = np.zeros(self.n_neurons)

        # Update membrane potentials
        non_refractory = self.refractory_time <= 0

        # Decay potential toward rest
        self.v[non_refractory] += dt * (-(self.v[non_refractory] - self.v_rest) +
                                         external_input[non_refractory]) / self.tau

        # Check for spikes
        spiked = (self.v >= self.v_threshold)

        # Record spikes
        for i in np.where(spiked)[0]:
            self.spike_times[i].append(self.current_time)

        # Reset membrane potential and set refractory period for spiked neurons
        self.v[spiked] = self.v_reset
        self.refractory_time[spiked] = 2.0  # 2ms refractory period

        # Decrement refractory time
        self.refractory_time -= dt

        # Add synaptic inputs from spiking neurons
        synaptic_input = np.dot(self.weights, spiked.astype(float))
        self.v[non_refractory] += synaptic_input[non_refractory]

        return spiked

    def run(self, duration, input_fn=None):
        """
        Run simulation for specified duration

        Parameters:
        - duration: Simulation duration (ms)
        - input_fn: Function that returns external input at each time step

        Returns:
        - spike_times: List of spike times for each neuron
        """
        steps = int(duration / 0.1)  # Assuming dt=0.1

        for step in range(steps):
            t = step * 0.1

            # Get external input if provided
            external_input = None
            if input_fn is not None:
                external_input = input_fn(t)
            self.step(external_input)

        return self.spike_times

    def plot_raster(self, neuron_subset=None, time_range=None):
        """
        Plot spike raster for simulated neurons

        Parameters:
        - neuron_subset: List of neuron indices to plot (default: first 100)
        - time_range: Time range to plot as [start, end] (default: all)
        """
        import matplotlib.pyplot as plt
        import numpy as np

        # Default to plotting first 100 neurons
        if neuron_subset is None:
            neuron_subset = range(min(100, self.n_neurons))

        # Create figure
        plt.figure(figsize=(12, 8))

        # Plot spikes for each neuron
        for i, neuron_idx in enumerate(neuron_subset):
            spikes = self.spike_times[neuron_idx]
            if time_range:
                spikes = [t for t in spikes if time_range[0] <= t <= time_range[1]]
            plt.scatter(spikes, np.ones_like(spikes) * i, marker='|', color='black',

        plt.xlabel('Time (ms)')
        plt.ylabel('Neuron index')
        plt.title('Spike Raster Plot')

        if time_range:
            plt.xlim(time_range)

        plt.tight_layout()
        plt.show()
```

Large-scale simulations enable:

- Testing hypotheses about neural circuit function
- Exploring emergent dynamics in complex networks
- Investigating how network structure shapes activity patterns
- Simulating the effects of interventions like stimulation or pharmacology

21.2.2 Brain Region Models

AI is helping develop increasingly detailed models of specific brain regions:

```python
def create_cortical_column_model(n_layers=6, neurons_per_layer=100):
    """
    Create a simplified model of a cortical column

    Parameters:
    - n_layers: Number of cortical layers
    - neurons_per_layer: Neurons per layer

    Returns:
    - model: Cortical column simulation model
    """
    import numpy as np

    n_neurons = n_layers * neurons_per_layer
    model = BrainRegionSimulation(n_neurons=n_neurons, connectivity_density=0.15)

    # Modify connectivity to reflect cortical architecture
    # Reset weights
    model.weights = np.zeros((n_neurons, n_neurons))

    # Layer-specific connection probabilities
    connection_probs = {
        # (pre_layer, post_layer): probability
        (0, 0): 0.15,  # Layer 1 → Layer 1
        (0, 1): 0.2,   # Layer 1 → Layer 2/3
        (1, 1): 0.2,   # Layer 2/3 → Layer 2/3
        (1, 2): 0.15,  # Layer 2/3 → Layer 4
        (1, 3): 0.1,   # Layer 2/3 → Layer 5
        (2, 1): 0.05,  # Layer 4 → Layer 2/3
        (2, 2): 0.15,  # Layer 4 → Layer 4
        (2, 3): 0.2,   # Layer 4 → Layer 5
        (3, 3): 0.15,  # Layer 5 → Layer 5
        (3, 4): 0.2,   # Layer 5 → Layer 6
        (3, 1): 0.05,  # Layer 5 → Layer 2/3 (feedback)
        (4, 2): 0.05,  # Layer 6 → Layer 4
        (4, 4): 0.1,   # Layer 6 → Layer 6
    }

    # Create connections based on probabilities
    for (pre_layer, post_layer), prob in connection_probs.items():
        pre_start = pre_layer * neurons_per_layer
        pre_end = (pre_layer + 1) * neurons_per_layer
        post_start = post_layer * neurons_per_layer
        post_end = (post_layer + 1) * neurons_per_layer

        # Random connectivity based on probability
        for i in range(post_start, post_end):
            for j in range(pre_start, pre_end):
                if np.random.random() < prob:
                    # Excitatory or inhibitory based on pre-synaptic neuron
                    is_inhibitory = (j % 5 == 0)  # ~20% inhibitory neurons
                    weight = np.random.normal(-0.5, 0.1) if is_inhibitory else np.ra
                    model.weights[i, j] = weight
    # Create thalamic input function
    def thalamic_input(t, target_layer=2):
        """Simulate periodic thalamic input to layer 4"""
        input_vec = np.zeros(n_neurons)

        # Target mainly layer 4 (index 2)
        target_start = target_layer * neurons_per_layer
        target_end = (target_layer + 1) * neurons_per_layer

        # Periodic input (every 100ms)
        if t % 100 < 5:
            # Random subset of neurons in target layer
            target_neurons = np.random.choice(
                range(target_start, target_end),
                size=int(neurons_per_layer * 0.2),
                replace=False
            )
            input_vec[target_neurons] = np.random.normal(1.0, 0.2, len(target_neuror

        return input_vec

    # Store the input function with the model
    model.thalamic_input = thalamic_input

    return model
```

Brain region models enable:

- **Structure-Function Analysis**: Relating anatomical organization to functional properties
- **Inter-Region Integration**: Understanding how different brain areas interact
- **Hierarchical Processing**: Modeling information flow through layered structures
- **Computational Comparisons**: Relating brain regions to artificial neural network architectures

21.3 Connectome Reconstruction

AI techniques are dramatically accelerating efforts to map the brain's wiring diagram (connectome) at multiple scales.

21.3.1 Electron Microscopy Analysis

Electron microscopy (EM) allows imaging of neural tissue at nanometer resolution, capturing synaptic connections. AI is essential for analyzing the enormous datasets this produces:

```python
def segment_neural_images(electron_microscopy_images):
    """
    Segment neurons in electron microscopy images using deep learning

    Parameters:
    - electron_microscopy_images: 3D stack of EM images

    Returns:
    - segmentation: 3D segmentation map
    """
    import torch
    import torch.nn as nn
    import numpy as np

    # Create a 3D U-Net model for segmentation
    class UNet3D(nn.Module):
        def __init__(self, in_channels=1, out_channels=3):
            super(UNet3D, self).__init__()
            # Simplified placeholder for the model architecture
            self.encoder = nn.Conv3d(in_channels, 16, kernel_size=3, padding=1)
            self.decoder = nn.Conv3d(16, out_channels, kernel_size=3, padding=1)

        def forward(self, x):
            # Simplified forward pass
            x = torch.relu(self.encoder(x))
            x = self.decoder(x)
            return x

    model = UNet3D(in_channels=1, out_channels=3)  # 3 output channels: background,

    # Process image stack in 3D patches
    patch_size = (64, 64, 64)
    segmentation = np.zeros_like(electron_microscopy_images)

    # Simplified inference (in practice, would need proper patch handling)
    for z in range(0, electron_microscopy_images.shape[0], patch_size[0]//2):
        for y in range(0, electron_microscopy_images.shape[1], patch_size[1]//2):
            for x in range(0, electron_microscopy_images.shape[2], patch_size[2]//2)
                # Extract patch
                z_end = min(z + patch_size[0], electron_microscopy_images.shape[0])
                y_end = min(y + patch_size[1], electron_microscopy_images.shape[1])
                x_end = min(x + patch_size[2], electron_microscopy_images.shape[2])

                patch = electron_microscopy_images[z:z_end, y:y_end, x:x_end]

                # Zero-pad if necessary
                if patch.shape != patch_size:
                    padded = np.zeros(patch_size)
                    padded[:patch.shape[0], :patch.shape[1], :patch.shape[2]] = patc
                    patch = padded

                # Predict segmentation
                with torch.no_grad():
```

```
                input_tensor = torch.tensor(patch, dtype=torch.float32).unsqueez
                prediction = model(input_tensor).argmax(dim=1).squeeze().numpy()

            # Update segmentation (handle overlap with averaging)
            segmentation[z:z_end, y:y_end, x:x_end] = prediction[:z_end-z, :y_er

    # Post-process to get instance segmentation (simplified)
    from skimage.measure import label
    instance_seg = label(segmentation == 2)  # Assuming channel 2 is cell interior

    return instance_seg
```

AI-powered connectome reconstruction enables:

- Automated segmentation of neurons in EM volumes
- Tracing of neurites through complex tissue samples
- Identification of synaptic connections
- Statistical analysis of connectivity patterns

21.3.2 Macro-Scale Connectomics

AI also helps analyze macro-scale connectivity using techniques like diffusion MRI:

```python
def analyze_structural_connectivity(dti_data, parcellation):
    """
    Analyze structural connectivity from diffusion MRI data

    Parameters:
    - dti_data: Diffusion tensor imaging data
    - parcellation: Brain region parcellation

    Returns:
    - connectivity_matrix: Structural connectivity matrix
    """
    import numpy as np
    import dipy.tracking.utils as utils
    from dipy.tracking.streamline import Streamlines

    # Placeholder for actual tractography implementation
    # In practice, this would use proper DTI processing tools

    # Simulate streamlines
    n_streamlines = 10000
    n_regions = len(np.unique(parcellation)) - 1  # Excluding background

    # Create random streamlines for demonstration
    streamlines = []
    for _ in range(n_streamlines):
        # Random streamline with 100 points
        streamline = np.random.rand(100, 3) * np.array(dti_data.shape)
        streamlines.append(streamline)

    # Convert to DIPY Streamlines object
    streamlines = Streamlines(streamlines)

    # Compute connectivity matrix (region x region)
    M, grouping = utils.connectivity_matrix(
        streamlines, parcellation,
        return_mapping=True,
        mapping_as_streamlines=True
    )

    # Normalize by region size
    region_sizes = np.bincount(parcellation.flat)[1:]  # Exclude background
    for i in range(n_regions):
        for j in range(n_regions):
            if region_sizes[i] > 0 and region_sizes[j] > 0:
                M[i, j] = M[i, j] / np.sqrt(region_sizes[i] * region_sizes[j])

    return M
```

Macro-scale connectome analysis reveals:

- **Brain network topology**: The large-scale organization of brain networks

- **Structural connectivity patterns**: How different brain regions connect to each other
- **Individual differences**: How connectivity varies across subjects or clinical populations
- **Structure-function relationships**: How structural connectivity constrains functional dynamics

21.4 Theory Development through Modeling

Beyond data analysis, AI helps develop computational theories of brain function by building models that explain neural data.

21.4.1 Computational Models of Cognition

Bayesian and probabilistic models can capture how the brain performs inference under uncertainty:

```
class BayesianInferenceBrain:
    def __init__(self, sensory_noise=0.1, prior_mean=0, prior_var=1.0):
        """
        Model of Bayesian inference in the brain

        Parameters:
        - sensory_noise: Standard deviation of sensory noise
        - prior_mean: Prior belief about the mean of the variable
        - prior_var: Prior belief about the variance of the variable
        """
        self.sensory_noise = sensory_noise
        self.prior_mean = prior_mean
        self.prior_var = prior_var

        # Current belief
        self.belief_mean = prior_mean
        self.belief_var = prior_var

    def update_belief(self, observation):
        """
        Update beliefs using Bayes' rule

        Parameters:
        - observation: New sensory observation

        Returns:
        - posterior_mean: Updated belief mean
        - posterior_var: Updated belief variance
        """
        # Compute precision (inverse variance)
        prior_precision = 1.0 / self.belief_var
        obs_precision = 1.0 / (self.sensory_noise ** 2)

        # Bayesian update (for Gaussian variables)
        posterior_precision = prior_precision + obs_precision
        posterior_var = 1.0 / posterior_precision

        posterior_mean = posterior_var * (
            prior_precision * self.belief_mean +
            obs_precision * observation
        )

        # Update beliefs
        self.belief_mean = posterior_mean
        self.belief_var = posterior_var

        return posterior_mean, posterior_var

    def predict_observation(self, n_samples=1000):
        """
        Generate predicted observations based on current belief

        Parameters:
```

```
        - n_samples: Number of samples to generate

        Returns:
        - samples: Predicted observations
        """
        import numpy as np

        # Sample from current belief
        samples = np.random.normal(self.belief_mean, np.sqrt(self.belief_var), n_sam

        # Add sensory noise
        samples += np.random.normal(0, self.sensory_noise, n_samples)

        return samples
```

21.4.2 Neural Circuit Mechanisms

AI helps identify circuit mechanisms that explain observed neural activity patterns:

```
def infer_circuit_mechanisms(neural_activity, behaviors):
    """
    Infer underlying circuit mechanisms from neural recordings

    Parameters:
    - neural_activity: Neural activity recordings [neurons, time]
    - behaviors: Behavioral measurements [time, behaviors]

    Returns:
    - circuit_model: Inferred circuit model
    """
    import numpy as np
    import torch
    import torch.nn as nn
    import torch.optim as optim

    n_neurons = neural_activity.shape[0]
    n_behaviors = behaviors.shape[1]

    # Define a circuit model
    class CircuitModel(nn.Module):
        def __init__(self, n_neurons, n_latent, n_behaviors):
            super(CircuitModel, self).__init__()
            self.n_neurons = n_neurons
            self.n_latent = n_latent

            # Recurrent weights
            self.W_rec = nn.Parameter(torch.randn(n_neurons, n_neurons) * 0.1)

            # Input weights
            self.W_in = nn.Parameter(torch.randn(n_neurons, n_behaviors) * 0.1)

            # Readout weights
            self.W_out = nn.Parameter(torch.randn(n_behaviors, n_neurons) * 0.1)

            # Latent dynamics
            self.W_latent = nn.Parameter(torch.randn(n_latent, n_latent) * 0.1)
            self.W_latent_to_neurons = nn.Parameter(torch.randn(n_neurons, n_latent)
            self.W_neurons_to_latent = nn.Parameter(torch.randn(n_latent, n_neurons)

            # Biases
            self.b_neurons = nn.Parameter(torch.zeros(n_neurons))
            self.b_latent = nn.Parameter(torch.zeros(n_latent))

        def forward(self, x_t, r_t, z_t):
            """Single timestep update"""
            # Update latent state
            z_next = torch.tanh(
                torch.matmul(self.W_latent, z_t) +
                torch.matmul(self.W_neurons_to_latent, r_t) +
                self.b_latent
            )
            # Update neural activity
            r_next = torch.relu(
                torch.matmul(self.W_rec, r_t) +
                torch.matmul(self.W_in, x_t) +
                torch.matmul(self.W_latent_to_neurons, z_t) +
                self.b_neurons
            )

            # Generate output
            y_next = torch.matmul(self.W_out, r_next)

            return r_next, z_next, y_next

        def run_simulation(self, inputs, steps, initial_r=None, initial_z=None):
            """Run simulation for multiple timesteps"""
            if initial_r is None:
                initial_r = torch.zeros(self.n_neurons)
            if initial_z is None:
                initial_z = torch.zeros(self.n_latent)

            r_t = initial_r
            z_t = initial_z

            # Store activity
            r_history = [r_t]
            z_history = [z_t]
            y_history = []

            # Run simulation
            for t in range(steps):
                x_t = inputs[t] if t < len(inputs) else torch.zeros_like(inputs[0])
                r_t, z_t, y_t = self.forward(x_t, r_t, z_t)

                r_history.append(r_t)
                z_history.append(z_t)
                y_history.append(y_t)

            return torch.stack(r_history), torch.stack(z_history), torch.stack(y_his

    # Prepare data
    X = torch.tensor(behaviors, dtype=torch.float32)
    Y = torch.tensor(neural_activity.T, dtype=torch.float32)  # [time, neurons]

    # Create model
    n_latent = 10  # Number of latent variables
    model = CircuitModel(n_neurons, n_latent, n_behaviors)

    # Define loss function and optimizer
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    # Train model
    n_steps = 100
    for epoch in range(100):
        # Reset initial state
        r0 = torch.zeros(n_neurons)
        z0 = torch.zeros(n_latent)

        # Forward pass
        r_pred, z_pred, y_pred = model.run_simulation(X, n_steps, r0, z0)

        # Computer loss (compare predicted to actual neural activity)
        loss = criterion(r_pred[1:], Y[:n_steps])

        # Backward pass and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if epoch % 10 == 0:
            print(f"Epoch {epoch}, Loss: {loss.item():.4f}")

    return model
```

21.5 Neuromorphic Applications

AI-derived insights about the brain can be applied to develop new neuromorphic computing architectures.

21.5.1 Brain-Inspired Learning Rules

Learning algorithms inspired by neuroscience can be more efficient than backpropagation:

```python
def train_with_local_learning_rule(network, inputs, targets, epochs=100, learning_ra
    """
    Train a neural network using a local Hebbian-like learning rule

    Parameters:
    - network: Neural network to train
    - inputs: Training inputs
    - targets: Training targets
    - epochs: Number of training epochs
    - learning_rate: Learning rate

    Returns:
    - network: Trained network
    """
    import numpy as np

    # Simple 2-layer network
    W1 = network['W1']  # Input → Hidden weights
    W2 = network['W2']  # Hidden → Output weights

    # Activation function
    def sigmoid(x):
        return 1 / (1 + np.exp(-x))

    # Derivative of sigmoid
    def dsigmoid(y):
        return y * (1 - y)

    for epoch in range(epochs):
        total_error = 0

        for i in range(len(inputs)):
            # Forward pass
            x = inputs[i]
            target = targets[i]

            # Hidden layer
            hidden_in = np.dot(W1, x)
            hidden_out = sigmoid(hidden_in)

            # Output layer
            output_in = np.dot(W2, hidden_out)
            output = sigmoid(output_in)

            # Calculate error
            error = target - output
            total_error += np.sum(error**2)

            # Local learning rule (simplified)
            # Output layer: Error-modulated Hebbian
            dW2 = learning_rate * np.outer(error * dsigmoid(output), hidden_out)

            # Hidden layer: Simplified error projection
            hidden_error = np.dot(W2.T, error * dsigmoid(output))
            dW1 = learning_rate * np.outer(hidden_error * dsigmoid(hidden_out), x)

            # Update weights
            W1 += dW1
            W2 += dW2

        if epoch % 10 == 0:
            print(f"Epoch {epoch}, Error: {total_error:.4f}")

    network['W1'] = W1
    network['W2'] = W2

    return network
```

Local learning rules have several advantages:

- **Biological plausibility**: They better match how the brain learns
- **Energy efficiency**: They require less information to be transmitted
- **Hardware compatibility**: They're easier to implement in neuromorphic systems
- **Continual learning**: They can adapt to new data without catastrophic forgetting

21.6 Code Lab: Neural Data Analysis

Let's implement a complete pipeline for neural data analysis:

```python
def neural_data_analysis_pipeline(neural_data_file):
    """
    End-to-end pipeline for neural data analysis

    Parameters:
    - neural_data_file: Path to neural recording data file

    Returns:
    - results: Analysis results and visualizations
    """
    import numpy as np
    import matplotlib.pyplot as plt
    from sklearn.decomposition import PCA
    from sklearn.manifold import TSNE
    import torch
    import torch.nn as nn
    import torch.optim as optim

    # Load data (simulated for this example)
    # In practice, would load from file
    np.random.seed(42)
    n_neurons = 100
    n_timepoints = 1000
    n_trials = 50
    n_conditions = 5

    # Create simulated data
    # Shape: [trials, neurons, time]
    data = np.random.randn(n_trials, n_neurons, n_timepoints) * 0.1

    # Add condition-specific patterns
    conditions = np.random.randint(0, n_conditions, n_trials)
    for i, condition in enumerate(conditions):
        # Create condition-specific neural pattern
        pattern = np.zeros(n_neurons)
        active_neurons = np.random.choice(n_neurons, 20, replace=False)
        pattern[active_neurons] = np.random.randn(20) * 2

        # Add pattern to neural activity with temporal dynamics
        for t in range(n_timepoints):
            if 200 <= t < 700:  # Active during middle of trial
                scale = np.sin((t - 200) / 500 * np.pi) * 3  # Modulation
                data[i, :, t] += pattern * scale

    # Add trial-to-trial variability
    data += np.random.randn(n_trials, n_neurons, n_timepoints) * 0.2

    # Create behavior data (e.g., reaction times)
    behavior = np.zeros(n_trials)
    for i, condition in enumerate(conditions):
        # Condition affects reaction time
        behavior[i] = 0.5 + condition * 0.1 + np.random.randn() * 0.1
```

```
print(f"Data loaded: {data.shape} (trials, neurons, time)")

# 1. Basic neural analysis

# Calculate trial-averaged firing rates
condition_avg = np.zeros((n_conditions, n_neurons, n_timepoints))
for c in range(n_conditions):
    condition_idx = conditions == c
    condition_avg[c] = data[condition_idx].mean(axis=0)

# Plot average firing rates for selected neurons
plt.figure(figsize=(12, 8))
selected_neurons = np.random.choice(n_neurons, 5, replace=False)
for i, neuron_idx in enumerate(selected_neurons):
    plt.subplot(5, 1, i+1)
    for c in range(n_conditions):
        plt.plot(condition_avg[c, neuron_idx], label=f"Condition {c}")
    plt.ylabel(f"Neuron {neuron_idx}")
    if i == 0:
        plt.title("Trial-averaged firing rates")
    if i == 4:
        plt.xlabel("Time (ms)")
        plt.legend()

plt.tight_layout()
plt.savefig("firing_rates.png")

# 2. Dimensionality reduction

# Reshape data for PCA [trials*time, neurons]
X = data.reshape(-1, n_neurons)

# Run PCA
pca = PCA(n_components=10)
X_pca = pca.fit_transform(X)

# Reshape back to [trials, time, components]
X_pca = X_pca.reshape(n_trials, n_timepoints, 10)

# Plot top 3 PCs for different conditions
plt.figure(figsize=(10, 8))
for c in range(n_conditions):
    condition_idx = conditions == c
    X_c = X_pca[condition_idx].mean(axis=0)  # Average across trials
    plt.subplot(5, 1, c+1)
    for pc in range(3):
        plt.plot(X_c[:, pc], label=f"PC{pc+1}")
    plt.title(f"Condition {c} - Top 3 PCs")
    plt.ylabel("PC Value")
    if c == 0:
        plt.legend()
    if c == 4:
        plt.xlabel("Time (ms)")
plt.tight_layout()
plt.savefig("pca_analysis.png")

# 3. Neural decoding

# Prepare data for decoding (use middle timepoint activity)
middle_idx = n_timepoints // 2
X_decode = data[:, :, middle_idx]  # [trials, neurons]
y_decode = conditions

# Split into train/test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X_decode, y_decode, test_size=0.3, random_state=42
)

# Simple decoder (multinomial logistic regression)
from sklearn.linear_model import LogisticRegression
decoder = LogisticRegression(multi_class='multinomial', solver='lbfgs')
decoder.fit(X_train, y_train)

# Evaluate
train_score = decoder.score(X_train, y_train)
test_score = decoder.score(X_test, y_test)
print(f"Decoding accuracy - Train: {train_score:.3f}, Test: {test_score:.3f}")

# Identify most informative neurons
coef = decoder.coef_  # [classes, features]
importance = np.abs(coef).sum(axis=0)  # Sum across classes
top_neurons = np.argsort(importance)[::-1][:10]  # Top 10 neurons

# 4. Dynamic neural trajectory visualization

# Use t-SNE to visualize neural dynamics
# Get trial-averaged activity for each condition
X_tsne = np.zeros((n_conditions * n_timepoints, n_neurons))
labels = np.zeros(n_conditions * n_timepoints, dtype=int)

for c in range(n_conditions):
    X_tsne[c * n_timepoints:(c+1) * n_timepoints] = condition_avg[c].T  # [time,
    labels[c * n_timepoints:(c+1) * n_timepoints] = c

# Run t-SNE
tsne = TSNE(n_components=2, random_state=42)
X_embedded = tsne.fit_transform(X_tsne)

# Plot trajectories
plt.figure(figsize=(10, 8))
colors = ['blue', 'green', 'red', 'purple', 'orange']

for c in range(n_conditions):
    idx = labels == c
    traj = X_embedded[idx]
    plt.plot(traj[:, 0], traj[:, 1], 'o-', color=colors[c], alpha=0.6,
             markersize=3, label=f"Condition {c}")

    # Mark start and end points
    plt.plot(traj[0, 0], traj[0, 1], 'x', color=colors[c], markersize=10)
    plt.plot(traj[-1, 0], traj[-1, 1], 's', color=colors[c], markersize=10)

plt.title("Neural Trajectories in t-SNE Space")
plt.legend()
plt.savefig("neural_trajectories.png")

# 5. Behavior prediction

# Try to predict behavior from neural activity
from sklearn.linear_model import Ridge

# Use average firing rates in the middle 200ms
mid_start, mid_end = n_timepoints//2 - 100, n_timepoints//2 + 100
X_behavior = data[:, :, mid_start:mid_end].mean(axis=2)  # [trials, neurons]

# Split data
X_train, X_test, y_train, y_test = train_test_split(
    X_behavior, behavior, test_size=0.3, random_state=42
)

# Train ridge regression
regressor = Ridge(alpha=1.0)
regressor.fit(X_train, y_train)

# Predict and evaluate
y_pred = regressor.predict(X_test)
from sklearn.metrics import r2_score
r2 = r2_score(y_test, y_pred)
print(f"Behavior prediction R² score: {r2:.3f}")

# Plot predictions vs actual
plt.figure(figsize=(8, 6))
plt.scatter(y_test, y_pred, alpha=0.6)
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], 'k--')
plt.xlabel("Actual Behavior")
plt.ylabel("Predicted Behavior")
plt.title(f"Neural Prediction of Behavior (R² = {r2:.3f})")
plt.savefig("behavior_prediction.png")

# Return results
results = {
    "n_neurons": n_neurons,
    "n_trials": n_trials,
    "n_conditions": n_conditions,
    "decoding_accuracy": test_score,
    "behavior_prediction_r2": r2,
    "top_neurons": top_neurons.tolist(),
    "pca_explained_variance": pca.explained_variance_ratio_.tolist()
}
return results
```

21.7 Take-aways

- **AI techniques are transforming neural data analysis**, enabling the extraction of meaningful patterns from complex recordings of brain activity.
- **Deep learning models make powerful decoders** that can predict behavior, perception, and cognitive states from neural activity.
- **Clinical neuroimaging applications** are leveraging AI for more accurate diagnosis and treatment of neurological disorders, enabling earlier detection of conditions like Alzheimer's disease and more precise surgical planning for epilepsy.
- **Multimodal integration of clinical data** with AI techniques provides comprehensive views of neurological conditions, combining structural, functional, genetic, and behavioral information for personalized medicine approaches.
- **Dimensionality reduction reveals low-dimensional neural manifolds** that capture the essential dynamics of neural population activity.
- **Computational simulations advance our understanding** of how neural circuits process information and generate behavior.
- **AI-powered connectome reconstruction** is mapping the brain's wiring diagram at multiple scales, from synapses to large-scale networks.
- **Neuromorphic computing applies brain-inspired principles** to create more efficient and adaptive AI systems.

21.8 Further Reading

- Richards, B.A., et al. (2019). A deep learning framework for neuroscience. Nature Neuroscience, 22(11), 1761-1770.
- Glaser, J.I., et al. (2020). Machine learning for neural decoding. eNeuro, 7(4).
- Pandarinath, C., et al. (2018). Inferring single-trial neural population dynamics using sequential auto-encoders. Nature Methods, 15(10), 805-815.
- Helmstaedter, M., et al. (2013). Connectomic reconstruction of the inner plexiform layer in the mouse retina. Nature, 500(7461), 168-174.

- Berman, G.J., et al. (2016). Predictability and hierarchy in Drosophila behavior. PNAS, 113(42), 11943-11948.