Chapter 19: Cognitive Neuroscience and Deep Learning

Chapter Goals

After completing this chapter, you will be able to:

- Understand the bidirectional relationship between cognitive neuroscience and deep learning
- Identify key cognitive neuroscience principles that have inspired deep learning architectures
- Apply cognitive constraints to improve deep learning model performance and interpretability
- Analyze deep learning models as computational models of cognition
- Explain how deep learning has contributed to our understanding of brain function
- Implement methods for comparing neural and artificial network representations
- Design experiments that bridge cognitive neuroscience and deep learning

19.1 Introduction: The Convergence of Minds and Machines

Cognitive neuroscience and deep learning represent two powerful approaches to understanding intelligence—one through the study of biological brains, and the other through the development of artificial neural systems. While these fields developed largely independently, they have begun to converge in recent years, creating a rich interdisciplinary area that promises to advance both our understanding of natural intelligence and our ability to create artificial intelligence.

This chapter explores this bidirectional relationship: how cognitive neuroscience inspires deep learning architectures and strategies, and how deep learning models serve as computational models of cognition that generate testable predictions about brain function.

```python
# Conceptual overview of the relationship between cognitive neuroscience and deep le
import matplotlib.pyplot as plt
import numpy as np
from matplotlib_venn import venn2

def visualize_field_relationship():
    """
    Visualize the bidirectional relationship between cognitive neuroscience and deep
    """
    fig, ax = plt.subplots(figsize=(10, 6))

    # Create a Venn diagram showing the overlap
    v = venn2(subsets=(0.4, 0.4, 0.2), set_labels=('Cognitive Neuroscience', 'Deep L

    # Change colors and alpha
    v.get_patch_by_id('10').set_color('lightblue')
    v.get_patch_by_id('01').set_color('lightgreen')
    v.get_patch_by_id('11').set_color('orange')

    v.get_patch_by_id('10').set_alpha(0.7)
    v.get_patch_by_id('01').set_alpha(0.7)
    v.get_patch_by_id('11').set_alpha(0.7)

    # Add bidirectional arrow to show mutual influence
    plt.annotate('', xy=(0.3, 0.6), xytext=(0.7, 0.6),
                 arrowprops=dict(arrowstyle='<->', color='black', lw=2))

    # Add examples of cross-disciplinary concepts in the overlap
    ax.text(0.5, 0.55, "Shared Concepts:", ha='center', fontweight='bold')
    ax.text(0.5, 0.5, "• Hierarchical processing", ha='center')
    ax.text(0.5, 0.45, "• Distributed representations", ha='center')
    ax.text(0.5, 0.4, "• Attention mechanisms", ha='center')
    ax.text(0.5, 0.35, "• Predictive coding", ha='center')

    # Add examples specific to each field
    ax.text(0.2, 0.7, "• Neural circuits", ha='center')
    ax.text(0.2, 0.65, "• Cognitive processes", ha='center')
    ax.text(0.2, 0.6, "• Brain imaging", ha='center')

    ax.text(0.8, 0.7, "• Backpropagation", ha='center')
    ax.text(0.8, 0.65, "• Gradient descent", ha='center')
    ax.text(0.8, 0.6, "• Layer architectures", ha='center')

    # Set title
    ax.set_title('The Bidirectional Relationship Between Cognitive Neuroscience and
                 fontsize=14, pad=20)

    plt.show()
```

19.2 Cognitive Science Principles in Deep Learning

19.2.1 Attention and Working Memory

The human attention system allows us to selectively focus on relevant information while filtering out distractions. In deep learning, attention mechanisms have revolutionized performance across domains:

- **Visual attention**: Mechanisms that weight the importance of different regions in an image
- **Self-attention**: Found in transformers, allows models to weigh the importance of different elements in a sequence
- **Cross-attention**: Allows models to relate elements from different modalities or sequences

Working memory—our ability to temporarily maintain and manipulate information—has also influenced deep learning through:

- **Memory networks**: Architectures with explicit memory components
- **Gating mechanisms**: Control the flow of information through the network
- **Meta-learning**: Learning to rapidly adapt to new tasks by maintaining task-relevant information

19.2.2 Hierarchical Processing and Compositionality

The brain processes information through hierarchical structures, from simple features to complex concepts. This principle has inspired deep learning architectures:

- **Convolutional neural networks**: Hierarchical visual processing from edges to objects
- **Hierarchical reinforcement learning**: Breaking complex tasks into manageable sub-goals
- **Compositional generalization**: Combining learned components in novel ways

```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, MaxPooling2D, Flatten

def create_hierarchical_cnn():
    """
    Create a CNN with hierarchical processing inspired by the visual cortex

    Returns:
    --------
    model : tf.keras.Model
        A CNN with hierarchical processing
    """
    model = Sequential([
        # Input layer (specify for clarity)
        tf.keras.layers.Input(shape=(224, 224, 3)),

        # Stage 1: Low-level feature extraction (analogous to V1)
        # Detect edges and simple contours
        Conv2D(32, (3, 3), activation='relu', padding='same', name='low_level_featur
        Conv2D(32, (3, 3), activation='relu', padding='same'),
        MaxPooling2D((2, 2)),

        # Stage 2: Mid-level feature extraction (analogous to V2/V4)
        # Detect shapes and textures
        Conv2D(64, (3, 3), activation='relu', padding='same', name='mid_level_featur
        Conv2D(64, (3, 3), activation='relu', padding='same'),
        MaxPooling2D((2, 2)),

        # Stage 3: Higher-level feature extraction (analogous to posterior IT)
        # Detect parts of objects
        Conv2D(128, (3, 3), activation='relu', padding='same', name='high_level_feat
        Conv2D(128, (3, 3), activation='relu', padding='same'),
        MaxPooling2D((2, 2)),

        # Stage 4: Object-level representation (analogous to anterior IT)
        # Detect whole objects
        Conv2D(256, (3, 3), activation='relu', padding='same', name='object_level_fe
        Conv2D(256, (3, 3), activation='relu', padding='same'),
        MaxPooling2D((2, 2)),

        # Flatten and dense layers (analogous to prefrontal cortex)
        # Abstract categorization and decision making
        Flatten(),
        Dense(512, activation='relu', name='abstract_features'),
        Dense(100, activation='softmax', name='classification')
    ])

    return model

def visualize_activations(model, example_image):
    """
    Visualize activations at different stages of the hierarchical network
```

```
    Parameters:
    -----------
    model : tf.keras.Model
        The hierarchical CNN model
    example_image : numpy.ndarray
        An example image to visualize activations for
    """
    # This would extract activations from different layers of the network
    # In a real implementation, you would use a model to extract these
    # Here we'll just sketch the concept

    layer_names = [
        'low_level_features',
        'mid_level_features',
        'high_level_features',
        'object_level_features'
    ]

    fig, axes = plt.subplots(1, len(layer_names), figsize=(15, 3))

    for i, layer_name in enumerate(layer_names):
        # In real code, this would extract actual activations
        # feature_map = get_layer_activation(model, layer_name, example_image)

        # For demonstration, we'll create mock activations
        if i == 0:  # Low-level (edges)
            feature_map = np.random.rand(56, 56)  # Simplified for visualization
        elif i == 1:  # Mid-level (textures)
            feature_map = np.random.rand(28, 28)
        elif i == 2:  # High-level (parts)
            feature_map = np.random.rand(14, 14)
        else:  # Object level
            feature_map = np.random.rand(7, 7)

        axes[i].imshow(feature_map, cmap='viridis')
        axes[i].set_title(f"{layer_name.replace('_', ' ').title()}")
        axes[i].axis('off')

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

19.2.3 Predictive Coding and Generative Models

A fundamental principle in cognitive neuroscience is that the brain continuously predicts future inputs, with perception arising from the integration of these predictions with sensory data. This has influenced deep learning through:

- **Generative models**: Systems that learn to generate likely inputs

- **Self-supervised learning**: Learning from prediction tasks without explicit labels
- **Contrastive predictive coding**: Learning representations by predicting future states
- **Variational autoencoders**: Learning latent representations that capture data distribution

19.2.4 Embodied Cognition and Active Learning

Cognitive science increasingly emphasizes that intelligence is embodied—developed through physical interaction with the environment. This has influenced AI through:

- **Reinforcement learning**: Agents learn from interactions with environments
- **Active learning**: Systems actively select what data to learn from
- **Curriculum learning**: Gradually increasing task difficulty during training
- **Curiosity-driven learning**: Using prediction errors to drive exploration

19.3 Cognitive Constraints in Deep Learning

19.3.1 Inductive Biases from Cognitive Science

Human cognition demonstrates numerous inductive biases—prior assumptions that guide learning. Incorporating these biases into deep learning models can improve performance:

- **Object-centric representations**: Humans naturally parse scenes into discrete objects
- **Causal reasoning**: Humans infer and reason about cause-and-effect relationships
- **Compositional structure**: Humans represent concepts as combinations of simpler parts
- **Few-shot learning**: Humans can learn from very few examples

```python
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F

class ObjectCentricNetwork(nn.Module):
    """
    A network that incorporates an object-centric inductive bias
    inspired by human visual cognition
    """

    def __init__(self, n_slots=5, slot_dim=64, hidden_dim=64):
        """
        Initialize the object-centric network

        Parameters:
        -----------
        n_slots : int
            Number of object slots
        slot_dim : int
            Dimension of each object slot
        hidden_dim : int
            Dimension of hidden layers
        """
        super().__init__()

        self.n_slots = n_slots
        self.slot_dim = slot_dim

        # Slot attention mechanism
        self.slot_attention = SlotAttention(
            dim=hidden_dim,
            n_slots=n_slots,
            slot_dim=slot_dim
        )

        # CNN encoder to extract features from images
        self.encoder = nn.Sequential(
            nn.Conv2d(3, 32, 5, stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 32, 5, stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, 5, stride=1, padding=2),
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(64 * 8 * 8, hidden_dim)
        )

        # Object-wise MLP for classification
        self.object_classifier = nn.Sequential(
            nn.Linear(slot_dim, hidden_dim),
```

```
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, 10)  # 10 object classes
        )

    def forward(self, x):
        """
        Forward pass through the network

        Parameters:
        -----------
        x : torch.Tensor
            Input images of shape (batch_size, channels, height, width)

        Returns:
        --------
        object_preds : torch.Tensor
            Object class predictions
        attention_maps : torch.Tensor
            Attention maps for each object slot
        """
        batch_size = x.shape[0]

        # Extract image features
        features = self.encoder(x)

        # Apply slot attention to segment into objects
        slots, attention_maps = self.slot_attention(features, x.shape)

        # Classify each object
        object_preds = self.object_classifier(slots.view(batch_size * self.n_slots,
        object_preds = object_preds.view(batch_size, self.n_slots, -1)

        return object_preds, attention_maps


class SlotAttention(nn.Module):
    """
    Simplified version of Slot Attention mechanism
    """

    def __init__(self, dim, n_slots, slot_dim):
        super().__init__()
        self.dim = dim
        self.n_slots = n_slots
        self.slot_dim = slot_dim

        # Initialize slot parameters
        self.slots = nn.Parameter(torch.randn(1, n_slots, slot_dim))

        # Projection layers
        self.to_q = nn.Linear(slot_dim, dim)
        self.to_k = nn.Linear(dim, dim)
```

```python
        self.to_v = nn.Linear(dim, dim)

        self.gru = nn.GRUCell(dim, slot_dim)

        # MLP for slot update
        self.mlp = nn.Sequential(
            nn.Linear(slot_dim, slot_dim),
            nn.ReLU(),
            nn.Linear(slot_dim, slot_dim)
        )

        # Layer norm
        self.norm_slots = nn.LayerNorm(slot_dim)
        self.norm_inputs = nn.LayerNorm(dim)

    def forward(self, inputs, image_shape, num_iterations=3):
        """
        Apply slot attention mechanism

        Parameters:
        -----------
        inputs : torch.Tensor
            Input features of shape (batch_size, dim)
        image_shape : tuple
            Shape of the input images (batch_size, channels, height, width)
        num_iterations : int
            Number of attention iterations

        Returns:
        --------
        slots : torch.Tensor
            Updated slot representations
        attention_maps : torch.Tensor
            Attention maps for visualization
        """
        batch_size = inputs.shape[0]

        # Initialize slots
        slots = self.slots.expand(batch_size, -1, -1)

        # Normalizations
        inputs = self.norm_inputs(inputs)

        # Reshape inputs for attention visualization
        h, w = image_shape[2] // 4, image_shape[3] // 4  # Downsampled due to CNN

        # Multiple rounds of attention
        for _ in range(num_iterations):
            slots_prev = slots

            # Normalize slots
            slots = self.norm_slots(slots)

            # Attention
```

```
            q = self.to_q(slots)
            k = self.to_k(inputs.unsqueeze(1))
            v = self.to_v(inputs.unsqueeze(1))

            # Compute attention scores
            attn_logits = torch.sum(q.unsqueeze(2) * k, dim=-1)
            attn = F.softmax(attn_logits, dim=1)

            # Weight values by attention
            updates = torch.sum(attn.unsqueeze(-1) * v, dim=2)

            # Update slots
            slots = self.gru(
                updates.reshape(-1, self.dim),
                slots_prev.reshape(-1, self.slot_dim)
            )
            slots = slots.reshape(batch_size, self.n_slots, self.slot_dim)

            # MLP for additional processing
            slots = slots + self.mlp(slots)

        # Reshape attention for visualization
        attention_maps = attn.reshape(batch_size, self.n_slots, h, w)

        return slots, attention_maps
```

19.3.2 Neural Architecture Constraints

Beyond specific cognitive principles, the broader architecture of the brain can inform deep learning:

- **Computational resource constraints**: Optimizing for energy efficiency
- **Local learning rules**: Alternatives to global backpropagation
- **Modular architectures**: Specialized components with distinct functions
- **Recurrence and feedback connections**: Incorporating temporal dynamics and top-down processing

19.3.3 Cognitively-Plausible Learning Mechanisms

Human learning differs from standard deep learning approaches in key ways:

- **Hebbian learning**: Connections strengthen when neurons co-activate
- **Contrastive learning**: Learning from differences between positive and negative examples

- **Curriculum learning**: Gradually increasing task difficulty
- **Few-shot and continual learning**: Learning efficiently from limited data while avoiding catastrophic forgetting

```python
import numpy as np

class HebbianNetwork:
    """
    Simple implementation of a Hebbian learning network
    """

    def __init__(self, input_size, output_size, learning_rate=0.01, decay_rate=0.000
        """
        Initialize the Hebbian network

        Parameters:
        -----------
        input_size : int
            Size of input features
        output_size : int
            Size of output features
        learning_rate : float
            Learning rate for weight updates
        decay_rate : float
            Weight decay rate to prevent unbounded growth
        """
        self.weights = np.random.normal(0, 0.1, (output_size, input_size))
        self.learning_rate = learning_rate
        self.decay_rate = decay_rate

    def forward(self, x):
        """
        Forward pass through the network

        Parameters:
        -----------
        x : numpy.ndarray
            Input data of shape (batch_size, input_size)

        Returns:
        --------
        y : numpy.ndarray
            Output activations of shape (batch_size, output_size)
        """
        return np.dot(x, self.weights.T)

    def update(self, x, y):
        """
        Update weights using Hebbian learning rule:
        "Neurons that fire together, wire together"

        Parameters:
        -----------
        x : numpy.ndarray
            Input data of shape (batch_size, input_size)
        y : numpy.ndarray
            Output activations of shape (batch_size, output_size)
```

```
        """
        # Basic Hebbian update
        delta_w = self.learning_rate * np.dot(y.T, x)

        # Apply weight decay to prevent unbounded growth
        delta_w -= self.decay_rate * self.weights

        # Update weights
        self.weights += delta_w

    def train(self, x, num_epochs=1):
        """
        Train the network for a specified number of epochs

        Parameters:
        -----------
        x : numpy.ndarray
            Input data of shape (batch_size, input_size)
        num_epochs : int
            Number of training epochs
        """
        for epoch in range(num_epochs):
            # Forward pass
            y = self.forward(x)

            # Update weights
            self.update(x, y)

            # Optional: Apply normalization to stabilize learning
            self.weights = self.weights / np.maximum(np.linalg.norm(self.weights, ax

            # Print progress
            if epoch % 10 == 0:
                print(f"Epoch {epoch}: Average activation: {np.mean(np.abs(y)):.4f}'

    def visualize_weights(self, reshape=None):
        """
        Visualize the learned weights

        Parameters:
        -----------
        reshape : tuple or None
            Reshape dimensions for visualizing weights as images
        """
        fig, axes = plt.subplots(1, min(5, self.weights.shape[0]), figsize=(15, 3))

        for i, ax in enumerate(axes):
            if i < self.weights.shape[0]:
                weight = self.weights[i]

                if reshape is not None:
                    weight = weight.reshape(reshape)

                ax.imshow(weight, cmap='viridis')
```

```
            ax.set_title(f"Neuron {i+1}")
            ax.axis('off')

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

19.4 Deep Learning Models as Theories of Cognition

19.4.1 Using Deep Learning to Test Cognitive Theories

Deep learning models can serve as computational implementations of cognitive theories:

- **Explicit formalizations**: Converting verbal theories into precise computations
- **Parameter exploration**: Testing hypotheses by manipulating model parameters
- **Counterfactual testing**: Exploring alternative mechanisms
- **Developmental trajectories**: Studying how learning unfolds over time

19.4.2 Case Studies in Cognitive Modeling

Deep learning has been used to model various cognitive domains:

- **Visual perception**: CNNs as models of object recognition
- **Language processing**: Transformers as models of language comprehension
- **Decision making**: Reinforcement learning as models of value-based choice
- **Memory**: Sequence models as models of episodic and working memory

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import matplotlib.pyplot as plt

class VisualCognitiveModel(nn.Module):
    """
    CNN-based model of visual object recognition designed to test
    cognitive theories about human visual processing
    """

    def __init__(self, with_recurrence=False, with_feedback=False):
        """
        Initialize the visual cognitive model

        Parameters:
        -----------
        with_recurrence : bool
            Whether to include recurrent connections
        with_feedback : bool
            Whether to include feedback connections
        """
        super().__init__()

        self.with_recurrence = with_recurrence
        self.with_feedback = with_feedback

        # Feedforward pathway (V1-like)
        self.conv1 = nn.Conv2d(3, 32, kernel_size=5, padding=2)
        self.pool1 = nn.MaxPool2d(2)

        # Feedforward pathway (V2-like)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5, padding=2)
        self.pool2 = nn.MaxPool2d(2)

        # Feedforward pathway (V4-like)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.pool3 = nn.MaxPool2d(2)

        # Feedforward pathway (IT-like)
        self.conv4 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
        self.pool4 = nn.MaxPool2d(2)

        # Recurrent connections
        if with_recurrence:
            self.recurrent1 = nn.Conv2d(32, 32, kernel_size=3, padding=1)
            self.recurrent2 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
            self.recurrent3 = nn.Conv2d(128, 128, kernel_size=3, padding=1)

        # Feedback connections
        if with_feedback:
            self.feedback3 = nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, pa
```

```
        self.feedback2 = nn.ConvTranspose2d(64, 32, kernel_size=4, stride=2, pad

    # Readout layers
    self.flatten = nn.Flatten()
    self.fc1 = nn.Linear(256 * 4 * 4, 512)
    self.fc2 = nn.Linear(512, 10)  # 10 object classes

    # Save activations for visualization
    self.activations = {}

def forward(self, x, timesteps=3):
    """
    Forward pass through the network

    Parameters:
    -----------
    x : torch.Tensor
        Input images
    timesteps : int
        Number of timesteps for recurrent processing

    Returns:
    --------
    output : torch.Tensor
        Class predictions
    """
    batch_size = x.shape[0]

    # Initial feedforward pass
    x1 = F.relu(self.conv1(x))
    p1 = self.pool1(x1)

    x2 = F.relu(self.conv2(p1))
    p2 = self.pool2(x2)

    x3 = F.relu(self.conv3(p2))
    p3 = self.pool3(x3)

    x4 = F.relu(self.conv4(p3))
    p4 = self.pool4(x4)

    # Store initial activations
    self.activations['layer1'] = p1.detach().cpu().numpy()
    self.activations['layer2'] = p2.detach().cpu().numpy()
    self.activations['layer3'] = p3.detach().cpu().numpy()
    self.activations['layer4'] = p4.detach().cpu().numpy()

    # Recurrent and feedback processing
    if self.with_recurrence or self.with_feedback:
        for t in range(timesteps - 1):
            # Store activations for this timestep
            self.activations[f'layer1_t{t+1}'] = p1.detach().cpu().numpy()
            self.activations[f'layer2_t{t+1}'] = p2.detach().cpu().numpy()
            self.activations[f'layer3_t{t+1}'] = p3.detach().cpu().numpy()
```

```
            # Apply recurrent connections
            if self.with_recurrence:
                p1 = p1 + F.relu(self.recurrent1(p1))
                p2 = p2 + F.relu(self.recurrent2(p2))
                p3 = p3 + F.relu(self.recurrent3(p3))

            # Apply feedback connections
            if self.with_feedback:
                feedback_3to2 = self.feedback3(p3)
                feedback_2to1 = self.feedback2(p2)

                # Add feedback to earlier representations
                p2 = p2 + 0.2 * feedback_3to2
                p1 = p1 + 0.2 * feedback_2to1

                # Update forward pass with feedback influence
                x2 = F.relu(self.conv2(p1))
                p2 = self.pool2(x2)

                x3 = F.relu(self.conv3(p2))
                p3 = self.pool3(x3)

                x4 = F.relu(self.conv4(p3))
                p4 = self.pool4(x4)

    # Final classification
    flat = self.flatten(p4)
    fc1 = F.relu(self.fc1(flat))
    output = self.fc2(fc1)

    return output

def analyze_temporal_dynamics(self, image, target_class, timesteps=5):
    """
    Analyze how representation evolves over time due to
    recurrent and feedback processing

    Parameters:
    -----------
    image : torch.Tensor
        Input image
    target_class : int
        Target class for the image
    timesteps : int
        Number of timesteps to analyze
    """
    # Ensure the model is in evaluation mode
    self.eval()

    # Forward pass with multiple timesteps
    output = self.forward(image, timesteps=timesteps)

    # Get class probabilities
```

```
        probs = F.softmax(output, dim=1)
        target_prob = probs[0, target_class].item()

        # Visualize how representations change over time
        fig, axes = plt.subplots(timesteps, 4, figsize=(15, 3*timesteps))

        for t in range(timesteps):
            for l in range(4):
                layer_name = f'layer{l+1}_t{t}' if t > 0 else f'layer{l+1}'
                if layer_name in self.activations:
                    # Take first image in batch, first channel for visualization
                    act = self.activations[layer_name][0, 0]
                    axes[t, l].imshow(act, cmap='viridis')
                    axes[t, l].set_title(f"Layer {l+1}, Time {t}")
                    axes[t, l].axis('off')

        plt.tight_layout()
        plt.show()

        # Plot target class probability over time
        plt.figure(figsize=(8, 4))
        plt.plot(range(timesteps), [probs[0, target_class].item() for t in range(tim
        plt.xlabel('Processing Timestep')
        plt.ylabel(f'Probability of Class {target_class}')
        plt.title('Temporal Dynamics of Recognition')
        plt.grid(True, alpha=0.3)
        plt.show()

        return probs
```

19.4.3 Comparing Model Behavior to Human Behavior

A key test of cognitive models is their ability to predict human behavior:

- **Psychophysical experiments**: Testing if models show similar perceptual biases
- **Error patterns**: Comparing model and human mistakes
- **Reaction times**: Relating model processing to response latencies
- **Developmental trajectories**: Comparing learning curves

19.5 Neural Representation Comparison Methods

19.5.1 Representational Similarity Analysis

Representational Similarity Analysis (RSA) is a framework for comparing neural representations across species, methods, and models:

- **Constructing similarity matrices**: Computing pairwise similarities between activity patterns
- **Computing representational similarity**: Correlating similarity matrices across systems
- **Significance testing**: Statistical approaches for assessing similarity
- **Visualization techniques**: Visualizing representational spaces

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import spearmanr, pearsonr
from scipy.spatial.distance import pdist, squareform

class RepresentationalSimilarityAnalysis:
    """
    Implementation of Representational Similarity Analysis (RSA)
    for comparing neural and model representations
    """

    def __init__(self, distance_metric='correlation'):
        """
        Initialize RSA

        Parameters:
        -----------
        distance_metric : str
            Distance metric for computing dissimilarity
            Options: 'correlation', 'euclidean', 'cosine'
        """
        self.distance_metric = distance_metric

    def compute_rdm(self, activations):
        """
        Compute Representational Dissimilarity Matrix (RDM)

        Parameters:
        -----------
        activations : numpy.ndarray
            Neural/model activations of shape (n_samples, n_features)

        Returns:
        --------
        rdm : numpy.ndarray
            Representational Dissimilarity Matrix of shape (n_samples, n_samples)
        """
        # Compute pairwise distances
        distances = pdist(activations, metric=self.distance_metric)

        # Convert to square form
        rdm = squareform(distances)

        return rdm

    def compare_rdms(self, rdm1, rdm2, method='spearman'):
        """
        Compare two RDMs to quantify representational similarity

        Parameters:
        -----------
        rdm1 : numpy.ndarray
            First RDM of shape (n_samples, n_samples)
```

```
        rdm2 : numpy.ndarray
            Second RDM of shape (n_samples, n_samples)
        method : str
            Correlation method ('spearman' or 'pearson')

        Returns:
        --------
        correlation : float
            Correlation coefficient between the two RDMs
        p_value : float
            p-value for the correlation
        """
        # Flatten the upper triangular part of the RDMs (excluding diagonal)
        triu_indices = np.triu_indices(rdm1.shape[0], k=1)
        rdm1_flat = rdm1[triu_indices]
        rdm2_flat = rdm2[triu_indices]

        # Compute correlation
        if method == 'spearman':
            correlation, p_value = spearmanr(rdm1_flat, rdm2_flat)
        elif method == 'pearson':
            correlation, p_value = pearsonr(rdm1_flat, rdm2_flat)
        else:
            raise ValueError("Method must be 'spearman' or 'pearson'")

        return correlation, p_value

    def visualize_rdm(self, rdm, labels=None, title='Representational Dissimilarity
        """
        Visualize a Representational Dissimilarity Matrix

        Parameters:
        -----------
        rdm : numpy.ndarray
            RDM of shape (n_samples, n_samples)
        labels : list or None
            Labels for the samples
        title : str
            Title for the plot
        """
        plt.figure(figsize=(10, 8))
        plt.imshow(rdm, cmap='viridis')
        plt.colorbar(label='Dissimilarity')
        plt.title(title)

        if labels is not None:
            plt.xticks(range(len(labels)), labels, rotation=90)
            plt.yticks(range(len(labels)), labels)

        plt.tight_layout()
        plt.show()

    def visualize_comparison(self, rdm1, rdm2, labels1='Model', labels2='Brain', tit
        """
```

```
        Visualize a comparison between two RDMs

        Parameters:
        -----------
        rdm1 : numpy.ndarray
            First RDM
        rdm2 : numpy.ndarray
            Second RDM
        labels1 : str
            Label for the first RDM
        labels2 : str
            Label for the second RDM
        title : str
            Title for the plot
        """
        # Flatten upper triangular part
        triu_indices = np.triu_indices(rdm1.shape[0], k=1)
        rdm1_flat = rdm1[triu_indices]
        rdm2_flat = rdm2[triu_indices]

        # Compute correlation
        correlation, p_value = self.compare_rdms(rdm1, rdm2)

        # Create scatter plot
        plt.figure(figsize=(8, 8))
        plt.scatter(rdm1_flat, rdm2_flat, alpha=0.5)
        plt.xlabel(f'{labels1} Dissimilarity')
        plt.ylabel(f'{labels2} Dissimilarity')
        plt.title(f'{title}\nCorrelation: {correlation:.3f} (p={p_value:.3g})')

        # Add regression line
        z = np.polyfit(rdm1_flat, rdm2_flat, 1)
        p = np.poly1d(z)
        plt.plot(np.linspace(min(rdm1_flat), max(rdm1_flat), 100),
                p(np.linspace(min(rdm1_flat), max(rdm1_flat), 100)),
                'r--', linewidth=2)

        plt.tight_layout()
        plt.show()

    def mds_visualization(self, rdm, labels=None, title='MDS Visualization'):
        """
        Visualize the representational space using Multi-Dimensional Scaling (MDS)

        Parameters:
        -----------
        rdm : numpy.ndarray
            RDM of shape (n_samples, n_samples)
        labels : list or None
            Labels for the samples
        title : str
            Title for the plot
        """
        from sklearn.manifold import MDS
```

```python
    # Create MDS model
    mds = MDS(n_components=2, dissimilarity='precomputed', random_state=42)

    # Fit MDS model to RDM
    points = mds.fit_transform(rdm)

    # Plot results
    plt.figure(figsize=(10, 8))
    plt.scatter(points[:, 0], points[:, 1], s=100)

    if labels is not None:
        for i, label in enumerate(labels):
            plt.annotate(label, (points[i, 0], points[i, 1]),
                         fontsize=12, ha='center', va='center')

    plt.title(title)
    plt.grid(alpha=0.3)
    plt.axis('equal')
    plt.tight_layout()
    plt.show()

    return points
```

19.5.2 Neural Encoding and Decoding Models

Neural encoding and decoding create direct mappings between brain activity and model representations:

- **Encoding models**: Predicting neural responses from model activations
- **Decoding models**: Predicting stimuli from neural responses
- **Cross-validated prediction**: Assessing generalization of encoding/decoding models
- **Feature importance analysis**: Identifying critical dimensions of the representation

19.5.3 Canonical Correlation Analysis

Canonical Correlation Analysis (CCA) finds shared dimensions between neural and model representations:

- **Canonical variates**: Identifying maximally correlated dimensions
- **Shared variance**: Quantifying overlap between representations
- **Dimensionality analysis**: Determining the number of meaningful shared dimensions

- **Stimulus associations**: Relating shared dimensions to stimulus properties

```python
import numpy as np
from sklearn.cross_decomposition import CCA
import matplotlib.pyplot as plt

class CanonicalCorrelationAnalyzer:
    """
    Implementation of Canonical Correlation Analysis (CCA)
    for comparing neural and model representations
    """

    def __init__(self, n_components=2):
        """
        Initialize CCA

        Parameters:
        -----------
        n_components : int
            Number of canonical components to extract
        """
        self.n_components = n_components
        self.cca = CCA(n_components=n_components)
        self.correlations = None

    def fit(self, X, Y):
        """
        Fit CCA on two sets of features

        Parameters:
        -----------
        X : numpy.ndarray
            First feature set (e.g., neural data) of shape (n_samples, n_features_X)
        Y : numpy.ndarray
            Second feature set (e.g., model data) of shape (n_samples, n_features_Y)
        """
        # Fit CCA
        self.cca.fit(X, Y)

        # Transform data to canonical space
        X_c, Y_c = self.cca.transform(X, Y)

        # Compute correlations between canonical variates
        self.correlations = np.array([np.corrcoef(X_c[:, i], Y_c[:, i])[0, 1]
                                      for i in range(self.n_components)])

        return X_c, Y_c

    def transform(self, X, Y):
        """
        Transform data to canonical space

        Parameters:
        -----------
        X : numpy.ndarray
```

```python
            First feature set
        Y : numpy.ndarray
            Second feature set

        Returns:
        --------
        X_c : numpy.ndarray
            First feature set in canonical space
        Y_c : numpy.ndarray
            Second feature set in canonical space
        """
        return self.cca.transform(X, Y)

    def visualize_correlations(self, labels=None):
        """
        Visualize canonical correlations

        Parameters:
        -----------
        labels : list or None
            Labels for the components
        """
        if self.correlations is None:
            raise ValueError("CCA must be fit before visualizing correlations")

        plt.figure(figsize=(10, 6))

        plt.bar(range(1, self.n_components + 1), self.correlations)
        plt.xlabel('Canonical Component')
        plt.ylabel('Correlation')
        plt.title('Canonical Correlations')

        if labels:
            plt.xticks(range(1, self.n_components + 1), labels)

        plt.grid(axis='y', alpha=0.3)
        plt.tight_layout()
        plt.show()

    def visualize_canonical_variates(self, X_c, Y_c, sample_labels=None):
        """
        Visualize the first two canonical variates

        Parameters:
        -----------
        X_c : numpy.ndarray
            First feature set in canonical space
        Y_c : numpy.ndarray
            Second feature set in canonical space
        sample_labels : list or None
            Labels for the samples
        """
        if X_c.shape[1] < 2 or Y_c.shape[1] < 2:
            raise ValueError("Need at least 2 components for visualization")
```

```
        fig, axes = plt.subplots(1, 2, figsize=(15, 6))

        # Plot first set of canonical variates
        axes[0].scatter(X_c[:, 0], X_c[:, 1], s=80, alpha=0.7)
        axes[0].set_xlabel('First Canonical Variate')
        axes[0].set_ylabel('Second Canonical Variate')
        axes[0].set_title('Neural Representation')
        axes[0].grid(alpha=0.3)

        # Plot second set of canonical variates
        axes[1].scatter(Y_c[:, 0], Y_c[:, 1], s=80, alpha=0.7)
        axes[1].set_xlabel('First Canonical Variate')
        axes[1].set_ylabel('Second Canonical Variate')
        axes[1].set_title('Model Representation')
        axes[1].grid(alpha=0.3)

        if sample_labels is not None:
            for i, label in enumerate(sample_labels):
                axes[0].annotate(label, (X_c[i, 0], X_c[i, 1]), fontsize=10)
                axes[1].annotate(label, (Y_c[i, 0], Y_c[i, 1]), fontsize=10)

        plt.tight_layout()
        plt.show()

    def correlation_significance(self, X, Y, n_permutations=1000, alpha=0.05):
        """
        Perform permutation test to assess significance of canonical correlations

        Parameters:
        -----------
        X : numpy.ndarray
            First feature set
        Y : numpy.ndarray
            Second feature set
        n_permutations : int
            Number of permutations for the test
        alpha : float
            Significance level

        Returns:
        --------
        p_values : numpy.ndarray
            p-values for each canonical correlation
        """
        if self.correlations is None:
            raise ValueError("CCA must be fit before testing significance")

        # Initialize array to store permutation correlations
        perm_correlations = np.zeros((n_permutations, self.n_components))

        # Original sample size
        n_samples = X.shape[0]
```

```
        # Perform permutation test
        for i in range(n_permutations):
            # Permute samples in Y
            perm_idx = np.random.permutation(n_samples)
            Y_perm = Y[perm_idx]

            # Fit CCA on permuted data
            cca_perm = CCA(n_components=self.n_components)
            cca_perm.fit(X, Y_perm)

            # Transform data to canonical space
            X_c_perm, Y_c_perm = cca_perm.transform(X, Y_perm)

            # Compute correlations
            for j in range(self.n_components):
                perm_correlations[i, j] = np.corrcoef(X_c_perm[:, j], Y_c_perm[:, j]

        # Compute p-values (proportion of permutation correlations >= observed)
        p_values = np.zeros(self.n_components)
        for j in range(self.n_components):
            p_values[j] = np.mean(perm_correlations[:, j] >= self.correlations[j])

        # Visualize results
        plt.figure(figsize=(12, 6))

        for j in range(self.n_components):
            plt.subplot(1, self.n_components, j+1)
            plt.hist(perm_correlations[:, j], bins=30, alpha=0.7, color='gray')
            plt.axvline(self.correlations[j], color='red', linestyle='--', linewidth
            plt.title(f'Component {j+1}: p={p_values[j]:.3f}')
            plt.xlabel('Correlation')
            plt.ylabel('Frequency')

        plt.tight_layout()
        plt.show()

        return p_values
```

19.6 The Impact of Deep Learning on Cognitive Neuroscience

19.6.1 New Frameworks for Understanding Brain Function

Deep learning has provided cognitive neuroscience with new conceptual tools:

- **Normative theories**: Explaining neural mechanisms as optimizations for specific objectives
- **Learning dynamics**: Understanding neural development through gradient-based learning
- **Distributed representations**: Conceptualizing neural coding as distributed patterns
- **End-to-end optimization**: Viewing brain regions as components in differentiable systems

19.6.2 Tools for Neural Data Analysis

Beyond conceptual advances, deep learning has provided practical tools for neuroscience:

- **Neural decoding**: Better extraction of information from brain recordings
- **Dimensionality reduction**: Discovering meaningful latent structures in neural data
- **Generative modeling**: Creating detailed models of neural activity
- **Automated analysis**: Processing and classifying large neural datasets

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import matplotlib.pyplot as plt

class LatentDynamicsModel(nn.Module):
    """
    Neural latent dynamics model for analyzing neural population activity
    """

    def __init__(self, n_neurons, latent_dim=3, nonlinearity='tanh'):
        """
        Initialize the latent dynamics model

        Parameters:
        -----------
        n_neurons : int
            Number of neurons in the population
        latent_dim : int
            Dimensionality of the latent space
        nonlinearity : str
            Nonlinearity to use ('relu', 'tanh', or 'sigmoid')
        """
        super().__init__()

        self.n_neurons = n_neurons
        self.latent_dim = latent_dim

        # Encoder: neural activity -> latent variables
        self.encoder = nn.Sequential(
            nn.Linear(n_neurons, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, latent_dim * 2)  # Mean and log-variance
        )

        # Latent dynamics model
        if nonlinearity == 'relu':
            self.dynamics_nonlinearity = nn.ReLU()
        elif nonlinearity == 'tanh':
            self.dynamics_nonlinearity = nn.Tanh()
        elif nonlinearity == 'sigmoid':
            self.dynamics_nonlinearity = nn.Sigmoid()
        else:
            raise ValueError("Nonlinearity must be 'relu', 'tanh', or 'sigmoid'")

        self.dynamics = nn.Sequential(
            nn.Linear(latent_dim, latent_dim),
            self.dynamics_nonlinearity
        )
```

```
        # Decoder: latent variables -> neural activity
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, 64),
            nn.ReLU(),
            nn.Linear(64, 128),
            nn.ReLU(),
            nn.Linear(128, n_neurons)
        )

    def encode(self, x):
        """
        Encode neural activity to latent variables

        Parameters:
        -----------
        x : torch.Tensor
            Neural activity of shape (batch_size, n_neurons)

        Returns:
        --------
        mean : torch.Tensor
            Mean of latent distribution
        logvar : torch.Tensor
            Log-variance of latent distribution
        """
        h = self.encoder(x)
        mean, logvar = torch.chunk(h, 2, dim=1)
        return mean, logvar

    def reparameterize(self, mean, logvar):
        """
        Reparameterization trick for sampling from latent distribution

        Parameters:
        -----------
        mean : torch.Tensor
            Mean of latent distribution
        logvar : torch.Tensor
            Log-variance of latent distribution

        Returns:
        --------
        z : torch.Tensor
            Sampled latent variables
        """
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mean + eps * std

    def decode(self, z):
        """
        Decode latent variables to neural activity

        Parameters:
```

```
        -----------
        z : torch.Tensor
            Latent variables of shape (batch_size, latent_dim)

        Returns:
        --------
        x_recon : torch.Tensor
            Reconstructed neural activity
        """
        return self.decoder(z)

    def forward(self, x):
        """
        Forward pass through the model

        Parameters:
        -----------
        x : torch.Tensor
            Neural activity of shape (batch_size, n_neurons)

        Returns:
        --------
        x_recon : torch.Tensor
            Reconstructed neural activity
        mean : torch.Tensor
            Mean of latent distribution
        logvar : torch.Tensor
            Log-variance of latent distribution
        z : torch.Tensor
            Sampled latent variables
        """
        mean, logvar = self.encode(x)
        z = self.reparameterize(mean, logvar)
        x_recon = self.decode(z)
        return x_recon, mean, logvar, z

    def predict_next_state(self, x):
        """
        Predict the next state in the latent space

        Parameters:
        -----------
        x : torch.Tensor
            Current neural activity

        Returns:
        --------
        x_next : torch.Tensor
            Predicted next neural activity
        """
        mean, _ = self.encode(x)
        z_next = self.dynamics(mean)
        x_next = self.decode(z_next)
        return x_next
```

```
    def loss_function(self, x_recon, x, mean, logvar, beta=1.0):
        """
        Compute the VAE loss function

        Parameters:
        -----------
        x_recon : torch.Tensor
            Reconstructed neural activity
        x : torch.Tensor
            Original neural activity
        mean : torch.Tensor
            Mean of latent distribution
        logvar : torch.Tensor
            Log-variance of latent distribution
        beta : float
            Weight of the KL divergence term

        Returns:
        --------
        loss : torch.Tensor
            Total loss
        recon_loss : torch.Tensor
            Reconstruction loss
        kl_loss : torch.Tensor
            KL divergence loss
        """
        # Reconstruction loss (mean squared error)
        recon_loss = F.mse_loss(x_recon, x, reduction='sum')

        # KL divergence loss
        kl_loss = -0.5 * torch.sum(1 + logvar - mean.pow(2) - logvar.exp())

        # Total loss
        loss = recon_loss + beta * kl_loss

        return loss, recon_loss, kl_loss

    def visualize_latent_space(self, data, labels=None, title='Latent Space Visualiz
        """
        Visualize the latent space

        Parameters:
        -----------
        data : torch.Tensor
            Neural activity data
        labels : numpy.ndarray or None
            Labels for color-coding points
        title : str
            Plot title
        """
        # Switch to evaluation mode
        self.eval()
```

```
        # Encode data to get latent representations
        with torch.no_grad():
            mean, _ = self.encode(data)
            z = mean.cpu().numpy()

        # Create scatter plot
        fig = plt.figure(figsize=(10, 8))

        if z.shape[1] >= 3:
            ax = fig.add_subplot(111, projection='3d')

            if labels is not None:
                for i, label in enumerate(np.unique(labels)):
                    idx = np.where(labels == label)[0]
                    ax.scatter(z[idx, 0], z[idx, 1], z[idx, 2], label=f'Class {label
            else:
                ax.scatter(z[:, 0], z[:, 1], z[:, 2], s=50, alpha=0.7)

            ax.set_xlabel('Latent Dim 1')
            ax.set_ylabel('Latent Dim 2')
            ax.set_zlabel('Latent Dim 3')

        else:
            ax = fig.add_subplot(111)

            if labels is not None:
                for i, label in enumerate(np.unique(labels)):
                    idx = np.where(labels == label)[0]
                    ax.scatter(z[idx, 0], z[idx, 1], label=f'Class {label}', s=50, a
            else:
                ax.scatter(z[:, 0], z[:, 1], s=50, alpha=0.7)

            ax.set_xlabel('Latent Dim 1')
            ax.set_ylabel('Latent Dim 2')

        ax.set_title(title)

        if labels is not None:
            ax.legend()

        plt.tight_layout()
        plt.show()

        return z

    def visualize_trajectory(self, data_sequence, title='Neural Trajectory in Latent
        """
        Visualize a neural trajectory in the latent space

        Parameters:
        -----------
        data_sequence : torch.Tensor
            Sequence of neural activity patterns
        title : str
```

```
            Plot title
        """
        # Switch to evaluation mode
        self.eval()

        # Encode sequence to get latent representations
        with torch.no_grad():
            latent_sequence = []
            for x in data_sequence:
                mean, _ = self.encode(x.unsqueeze(0))
                latent_sequence.append(mean.squeeze().cpu().numpy())

            latent_sequence = np.array(latent_sequence)

        # Create 3D plot for trajectory
        fig = plt.figure(figsize=(10, 8))

        if latent_sequence.shape[1] >= 3:
            ax = fig.add_subplot(111, projection='3d')

            # Plot trajectory
            ax.plot(latent_sequence[:, 0], latent_sequence[:, 1], latent_sequence[:,
                   'o-', linewidth=2, markersize=8)

            # Highlight start and end points
            ax.scatter(latent_sequence[0, 0], latent_sequence[0, 1], latent_sequence
                      color='green', s=100, label='Start')
            ax.scatter(latent_sequence[-1, 0], latent_sequence[-1, 1], latent_sequer
                      color='red', s=100, label='End')

            ax.set_xlabel('Latent Dim 1')
            ax.set_ylabel('Latent Dim 2')
            ax.set_zlabel('Latent Dim 3')

        else:
            ax = fig.add_subplot(111)

            # Plot trajectory
            ax.plot(latent_sequence[:, 0], latent_sequence[:, 1], 'o-',
                   linewidth=2, markersize=8)

            # Highlight start and end points
            ax.scatter(latent_sequence[0, 0], latent_sequence[0, 1],
                      color='green', s=100, label='Start')
            ax.scatter(latent_sequence[-1, 0], latent_sequence[-1, 1],
                      color='red', s=100, label='End')

            ax.set_xlabel('Latent Dim 1')
            ax.set_ylabel('Latent Dim 2')

        ax.set_title(title)
        ax.legend()
        plt.tight_layout()
        plt.show()
```

```python
        return latent_sequence
```

19.6.3 Generating New Hypotheses

Deep learning models can suggest novel hypotheses about brain function:

- **Optimization principles**: What objectives drive neural organization?
- **Architectural principles**: What network structures enable robust computation?
- **Learning mechanisms**: How does the brain learn efficiently from experience?
- **Feature representations**: What information is encoded in neural activity?

19.7 Practical Exercise: Comparing Deep Networks and Brain Representations

In this exercise, we'll demonstrate how to compare representations between a deep neural network and fMRI brain activity patterns in response to the same visual stimuli.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from scipy.stats import spearmanr

def simulate_fmri_data(n_stimuli=50, n_voxels=1000, n_roi=4, noise_level=0.1):
    """
    Simulate fMRI data for a visual experiment

    Parameters:
    -----------
    n_stimuli : int
        Number of stimuli (images)
    n_voxels : int
        Total number of voxels
    n_roi : int
        Number of brain regions (ROIs)
    noise_level : float
        Level of noise to add

    Returns:
    --------
    fmri_data : numpy.ndarray
        fMRI response patterns of shape (n_stimuli, n_voxels)
    roi_masks : list
        Binary masks for each ROI
    """
    # Create random fMRI patterns
    fmri_data = np.zeros((n_stimuli, n_voxels))

    # Create ROI masks (which voxels belong to which brain region)
    voxels_per_roi = n_voxels // n_roi
    roi_masks = []

    for r in range(n_roi):
        # Create binary mask for this ROI
        mask = np.zeros(n_voxels, dtype=bool)
        start_idx = r * voxels_per_roi
        end_idx = (r + 1) * voxels_per_roi if r < n_roi - 1 else n_voxels
        mask[start_idx:end_idx] = True
        roi_masks.append(mask)

        # Generate patterns for this ROI
        # We'll make each ROI sensitive to different stimulus features
        for i in range(n_stimuli):
            # Create a pattern that depends on stimulus index in different ways for
            if r == 0:  # Early visual (e.g., V1) - sensitive to low-level features
                pattern = np.sin(i / 5.0) + np.cos(i / 3.0)
            elif r == 1:  # Mid-level visual (e.g., V4) - sensitive to shapes
                pattern = np.sin(i / 8.0) * np.cos(i / 2.0)
            elif r == 2:  # Higher visual (e.g., LOC) - sensitive to objects
                pattern = np.tanh(i / 10.0 - 2.5)
```

```
            else:  # Object category (e.g., IT) - sensitive to categories
                pattern = (i % 5) / 5.0  # 5 categories

            fmri_data[i, mask] = pattern

    # Add noise
    fmri_data += noise_level * np.random.randn(*fmri_data.shape)

    return fmri_data, roi_masks

def simulate_cnn_activations(n_stimuli=50, n_layers=4, units_per_layer=250, noise_le
    """
    Simulate CNN activations for the same stimuli

    Parameters:
    -----------
    n_stimuli : int
        Number of stimuli (images)
    n_layers : int
        Number of CNN layers
    units_per_layer : int
        Number of units per layer
    noise_level : float
        Level of noise to add

    Returns:
    --------
    cnn_activations : dict
        Dictionary mapping layer names to activation patterns
    """
    cnn_activations = {}

    for l in range(n_layers):
        # Create activations for this layer
        layer_name = f"layer{l+1}"
        activations = np.zeros((n_stimuli, units_per_layer))

        for i in range(n_stimuli):
            # Create activations that depend on stimulus index in different ways for
            if l == 0:  # Conv1 - sensitive to edges
                pattern = np.sin(i / 5.0) + np.cos(i / 3.0)
            elif l == 1:  # Conv2 - sensitive to textures
                pattern = np.sin(i / 7.0) * np.cos(i / 2.0)
            elif l == 2:  # Conv3 - sensitive to parts
                pattern = np.sin(i / 10.0) * np.tanh(i / 4.0 - 2)
            else:  # Conv4 - sensitive to objects
                pattern = (i % 5) / 5.0  # 5 categories

            activations[i] = pattern + noise_level * np.random.randn(units_per_layer

        cnn_activations[layer_name] = activations

    return cnn_activations
```

```python
def compute_rdms(data_dict):
    """
    Compute Representational Dissimilarity Matrices (RDMs) for different regions/lay

    Parameters:
    -----------
    data_dict : dict
        Dictionary mapping region/layer names to activation patterns

    Returns:
    --------
    rdms : dict
        Dictionary mapping region/layer names to RDMs
    """
    rdms = {}

    for name, data in data_dict.items():
        # Compute pairwise correlation distances
        n_stimuli = data.shape[0]
        rdm = np.zeros((n_stimuli, n_stimuli))

        for i in range(n_stimuli):
            for j in range(i+1, n_stimuli):
                # 1 - correlation as a distance metric
                corr = np.corrcoef(data[i], data[j])[0, 1]
                dist = 1 - corr
                rdm[i, j] = dist
                rdm[j, i] = dist

        rdms[name] = rdm

    return rdms

def compare_representations(brain_rdms, model_rdms, roi_names=None, layer_names=None
    """
    Compare brain and model representations

    Parameters:
    -----------
    brain_rdms : dict
        Dictionary mapping ROI names to brain RDMs
    model_rdms : dict
        Dictionary mapping layer names to model RDMs
    roi_names : list or None
        Names of brain ROIs
    layer_names : list or None
        Names of model layers

    Returns:
    --------
    similarity_matrix : numpy.ndarray
        Matrix of correlations between brain ROIs and model layers
    """
    if roi_names is None:
```

```python
        roi_names = list(brain_rdms.keys())
    if layer_names is None:
        layer_names = list(model_rdms.keys())

    n_rois = len(roi_names)
    n_layers = len(layer_names)

    similarity_matrix = np.zeros((n_rois, n_layers))

    for i, roi in enumerate(roi_names):
        brain_rdm = brain_rdms[roi]

        # Extract upper triangular part (excluding diagonal)
        triu_indices = np.triu_indices_from(brain_rdm, k=1)
        brain_rdm_triu = brain_rdm[triu_indices]

        for j, layer in enumerate(layer_names):
            model_rdm = model_rdms[layer]
            model_rdm_triu = model_rdm[triu_indices]

            # Compute Spearman correlation between RDMs
            corr, _ = spearmanr(brain_rdm_triu, model_rdm_triu)
            similarity_matrix[i, j] = corr

    return similarity_matrix

def main():
    """
    Main function to run the analysis
    """
    # 1. Simulate brain fMRI data
    n_stimuli = 50
    print("Simulating fMRI data...")
    fmri_data, roi_masks = simulate_fmri_data(n_stimuli=n_stimuli)

    # 2. Create brain ROI data dictionary
    brain_data = {}
    roi_names = ["V1", "V4", "LOC", "IT"]

    for i, (name, mask) in enumerate(zip(roi_names, roi_masks)):
        brain_data[name] = fmri_data[:, mask]

    # 3. Simulate CNN activations
    print("Simulating CNN activations...")
    cnn_activations = simulate_cnn_activations(n_stimuli=n_stimuli)

    # 4. Compute RDMs for brain ROIs and CNN layers
    print("Computing RDMs...")
    brain_rdms = compute_rdms(brain_data)
    model_rdms = compute_rdms(cnn_activations)

    # 5. Compare representations
    print("Comparing representations...")
    similarity_matrix = compare_representations(brain_rdms, model_rdms, roi_names)
```

```python
# 6. Visualize results
layer_names = list(cnn_activations.keys())

plt.figure(figsize=(10, 8))
plt.imshow(similarity_matrix, cmap='viridis')
plt.colorbar(label='Representational Similarity (Spearman ρ)')
plt.xlabel('CNN Layers')
plt.ylabel('Brain ROIs')
plt.title('Brain-CNN Representational Similarity')
plt.xticks(range(len(layer_names)), layer_names)
plt.yticks(range(len(roi_names)), roi_names)

# Add text annotations
for i in range(len(roi_names)):
    for j in range(len(layer_names)):
        plt.text(j, i, f"{similarity_matrix[i, j]:.2f}",
                ha="center", va="center", color="white" if similarity_matrix[i,

plt.tight_layout()
plt.show()

# 7. Visualize RDMs
plt.figure(figsize=(15, 10))

# Plot brain RDMs
for i, name in enumerate(roi_names):
    plt.subplot(2, 4, i+1)
    plt.imshow(brain_rdms[name], cmap='viridis')
    plt.title(f"Brain: {name}")
    plt.colorbar(label='Dissimilarity')

# Plot model RDMs
for i, name in enumerate(layer_names):
    plt.subplot(2, 4, i+5)
    plt.imshow(model_rdms[name], cmap='viridis')
    plt.title(f"CNN: {name}")
    plt.colorbar(label='Dissimilarity')

plt.tight_layout()
plt.show()

# 8. Find the best matching layer for each ROI
best_layers = []
for i, roi in enumerate(roi_names):
    best_layer_idx = np.argmax(similarity_matrix[i])
    best_layer = layer_names[best_layer_idx]
    best_corr = similarity_matrix[i, best_layer_idx]
    best_layers.append((roi, best_layer, best_corr))

print("\nBest matching CNN layer for each brain ROI:")
for roi, layer, corr in best_layers:
    print(f"{roi}: {layer} (ρ = {corr:.3f})")
```

```python
if __name__ == "__main__":
    main()
```

19.8 Chapter Take-aways

- Cognitive neuroscience and deep learning have a bidirectional relationship, with each field informing the other
- Key cognitive principles like attention, hierarchical processing, and predictive coding have inspired advances in deep learning architectures
- Incorporating cognitive constraints and inductive biases can improve deep learning model performance and generalization
- Deep learning models serve as computational theories of cognition, generating testable predictions about brain function
- Methods like RSA, encoding models, and CCA enable direct comparisons between neural and artificial representations
- Deep learning has provided new frameworks and tools for understanding and analyzing brain function
- The convergence of these fields promises advances in both artificial intelligence and our understanding of human cognition

19.9 Further Reading

- Kriegeskorte, N., & Douglas, P. K. (2018). Cognitive computational neuroscience. *Nature Neuroscience, 21*(9), 1148-1160.
- Richards, B. A., et al. (2019). A deep learning framework for neuroscience. *Nature Neuroscience, 22*(11), 1761-1770.
- Yamins, D. L., & DiCarlo, J. J. (2016). Using goal-driven deep learning models to understand sensory cortex. *Nature Neuroscience, 19*(3), 356-365.
- Saxe, A., Nelli, S., & Summerfield, C. (2021). If deep learning is the answer, what is the question? *Nature Reviews Neuroscience, 22*(1), 55-67.
- Kietzmann, T. C., McClure, P., & Kriegeskorte, N. (2019). Deep neural networks in computational neuroscience. *Oxford Research Encyclopedia of Neuroscience*.

- Naselaris, T., et al. (2021). Cognitive computational neuroscience: A normative theory of brain function. *Nature Neuroscience, 24*(2), 291-306.