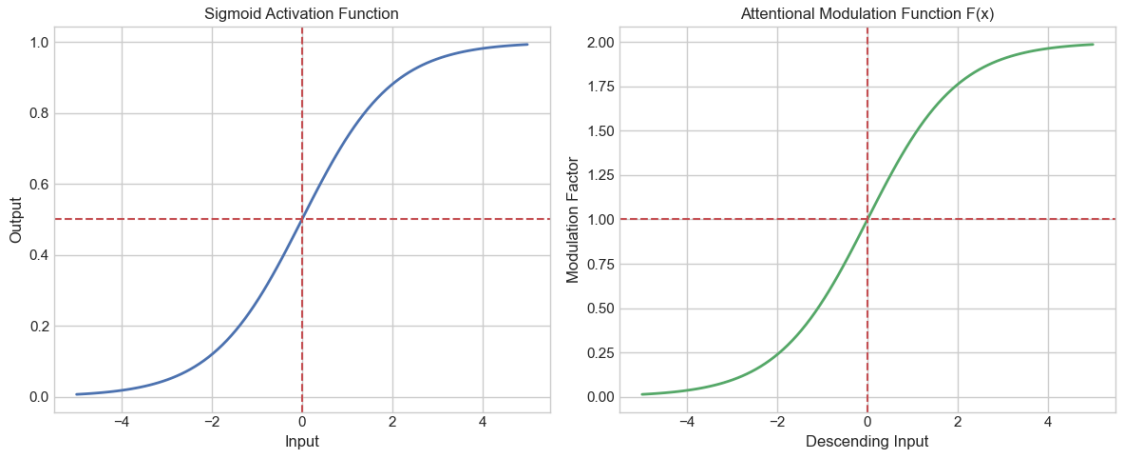
presentation

March 9, 2025

[6]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib.colors import LinearSegmentedColormap
from IPython.display import display, Markdown

sns.set(style="whitegrid")
plt.style.use("seaborn-v0_8-whitegrid")


# Define the sigmoid function as used in the paper
def sigmoid(x):
    return 1 / (1 + np.exp(-x))


# Define the modulating function F used for attentional gating
def modulating_function(x):
    # F(x) → 0 when x → -, F(0) = 1, and F(x) → 2 when x → +
    return 2 * sigmoid(x)


# Function to calculate inhibitory unit activity
def inhibitory_activation(weights, inputs):
    """
    S_INH = sigmoid( w_i,j S_j)
    """
    return sigmoid(np.sum(weights * inputs))


# Function to calculate excitatory unit activity with attention modulation
def excitatory_activation(asc_weights, asc_inputs, desc_weights, desc_inputs):
    """
    S_EXC = sigmoid(S_asc * F(S_desc))
    where S_asc =  w_i,j_asc S_j and S_desc =  w_i,j_desc S_j
    """
    S_asc = np.sum(asc_weights * asc_inputs)
    S_desc = np.sum(desc_weights * desc_inputs)
    return sigmoid(S_asc * modulating_function(S_desc))
```

```
# Reward-modulated Hebbian learning rule
def update_weights(weights, pre_activity, post_activity, reward, learning_rate=0.
↪1):
    """
    Δw_post,pre = ϵ * R * S_pre * (2*S_post - 1)
    R is reward signal (1 for correct, -1 for incorrect)
    """
    delta_w = learning_rate * reward * pre_activity * (2 * post_activity - 1)
    new_weights = weights + delta_w
    # Ensure weights stay within bounds (0 to 7 as mentioned in the paper)
    return np.clip(new_weights, 0, 7)


# Update rule for vigilance
def update_vigilance(vigilance, reward):
    """
    if R > 0, then ΔV = -0.1 V, otherwise ΔV = 0.5 (1 - V)
    """
    if reward > 0:
        delta_v = -0.1 * vigilance
    else:
        delta_v = 0.5 * (1 - vigilance)
    new_vigilance = vigilance + delta_v
    return np.clip(new_vigilance, 0, 1)  # Keep vigilance between 0 and 1


# Demo of the attentional gating mechanism
x = np.linspace(-5, 5, 100)
y_sigmoid = sigmoid(x)
y_modulation = modulating_function(x)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

ax1.plot(x, y_sigmoid, "b-", linewidth=2)
ax1.set_title("Sigmoid Activation Function")
ax1.set_xlabel("Input")
ax1.set_ylabel("Output")
ax1.grid(True)
ax1.axhline(y=0.5, color="r", linestyle="--")
ax1.axvline(x=0, color="r", linestyle="--")

ax2.plot(x, y_modulation, "g-", linewidth=2)
ax2.set_title("Attentional Modulation Function F(x)")
ax2.set_xlabel("Descending Input")
ax2.set_ylabel("Modulation Factor")
```

```
ax2.grid(True)
ax2.axhline(y=1, color="r", linestyle="--")
ax2.axvline(x=0, color="r", linestyle="--")

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

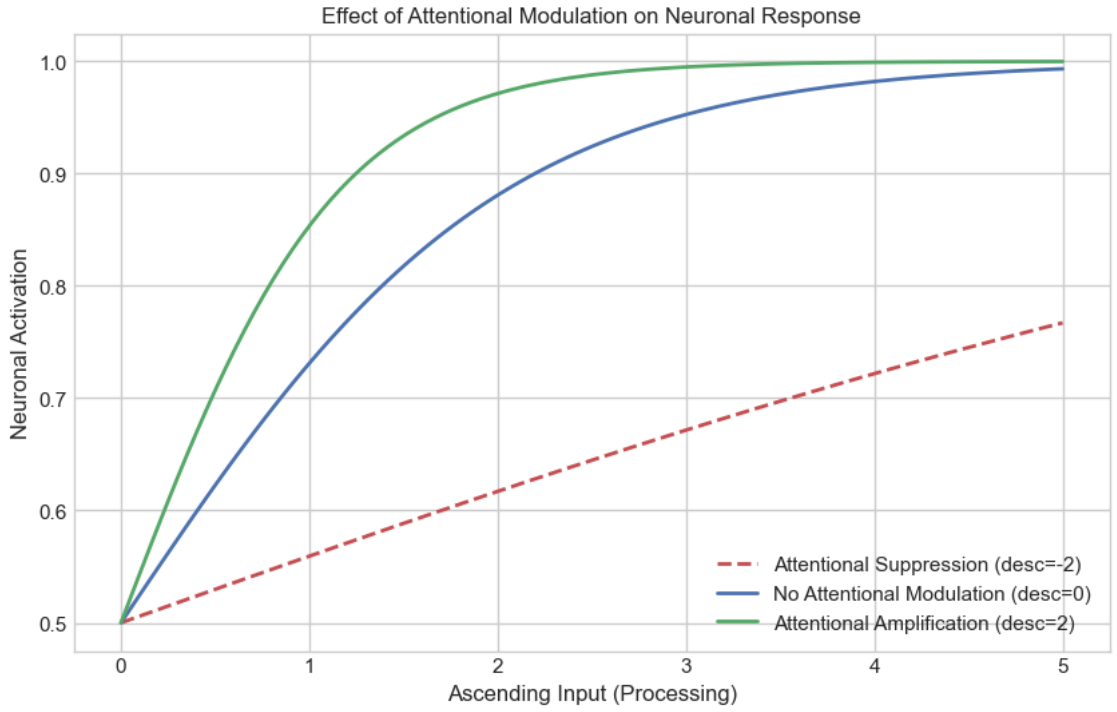


[2]:
```
display(
    Markdown(
        """
## Neuron Response to Attentional Modulation

The graph below shows how a neuron's response is affected by different levels of␣
↪ascending (regular) input and descending (attentional) input.
- When descending input is positive, it amplifies the response (attentional␣
↪amplification)
- When descending input is negative, it suppresses the response (attentional␣
↪suppression)
- When descending input is zero, the neuron responds normally to ascending input
"""
    )
)

# Demonstrate how attentional modulation affects neural activation
asc_inputs = np.linspace(0, 5, 100)  # Regular processing inputs
desc_inputs = np.array([-2, 0, 2])  # Attentional modulation (negative, none,␣
↪positive)

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

for desc in desc_inputs:
```

```
    activations = [sigmoid(asc * modulating_function(desc)) for asc in␣
↪asc_inputs]
    if desc < 0:
        label = f"Attentional Suppression (desc={desc})"
        style = "r--"
    elif desc > 0:
        label = f"Attentional Amplification (desc={desc})"
        style = "g-"
    else:
        label = f"No Attentional Modulation (desc={desc})"
        style = "b-"

    plt.plot(asc_inputs, activations, style, linewidth=2, label=label)

plt.title("Effect of Attentional Modulation on Neuronal Response")
plt.xlabel("Ascending Input (Processing)")
plt.ylabel("Neuronal Activation")
plt.legend()
plt.grid(True)
plt.show()
```

0.1 Neuron Response to Attentional Modulation

The graph below shows how a neuron's response is affected by different levels of ascending (regular) input and descending (attentional) input. - When descending input is positive, it amplifies the response (attentional amplification) - When descending input is negative, it suppresses the response (attentional suppression) - When descending input is zero, the neuron responds normally to ascending input



[4]:
```
display(
    Markdown(
        """
## Simulating Workspace Dynamics in the Stroop Task

The following simulation tracks the workspace activation during different phases␣
↪of learning the Stroop task.
We'll see how workspace activity changes across:
1. Initial routine tasks (no workspace needed)
2. Introduction of the effortful task
3. Search phase (variable workspace activation)
4. Effortful execution phase (stable workspace activation)
5. Error recovery (increased workspace activation)
6. Routinization (declining workspace activation)
"""
    )
)

# Simulate workspace dynamics during Stroop task learning
# This is a simplified approximation of Figure 3 in the paper

# Set parameters
n_trials = 200
n_workspace_units = 10  # Number of workspace neurons to visualize
```

```
# Initialize arrays to store simulation data
workspace_activity = np.zeros((n_workspace_units, n_trials))
mean_workspace = np.zeros(n_trials)
vigilance = np.ones(n_trials) * 0.5  # Initial vigilance level
errors = np.zeros(n_trials)
task_phases = np.zeros(n_trials)  # 0=routine, 1=search, 2=effortful,␣
↪3=routinization

# Task phase definitions
task_phases[:20] = 0  # Initial routine tasks
task_phases[20:50] = 1  # Search phase after Stroop introduction
task_phases[50:150] = 2  # Effortful execution
task_phases[150:] = 3  # Progressive routinization

# Set error patterns (simplification of the paper's Figure 3)
errors[20:25] = 1  # Initial errors when Stroop task is introduced
errors[30] = 1
errors[35] = 1
errors[40] = 1
errors[48] = 1  # Errors during search phase
errors[70] = 1
errors[90] = 1
errors[120] = 1
errors[170] = 1  # Occasional errors during routinization

# Define reward based on errors (inverted)
reward = 1 - 2 * errors  # +1 for correct, -1 for error

# Update vigilance based on reward
for i in range(1, n_trials):
    vigilance[i] = update_vigilance(vigilance[i - 1], reward[i - 1])

# Generate workspace activation patterns
# Phase 0: Routine tasks - minimal workspace activation
workspace_activity[:, :20] = np.random.uniform(0, 0.2, size=(n_workspace_units,␣
↪20))

# Phase 1: Search phase - variable workspace activation
for i in range(20, 50):
    if errors[i] == 1 and i > 20:  # After error, reset pattern
        workspace_activity[:, i] = np.random.uniform(0.5, 0.9, n_workspace_units)
    else:
        # Random exploration of workspace patterns
        workspace_activity[:, i] = np.random.uniform(0.3, 0.8, n_workspace_units)

    # Scale by vigilance
```

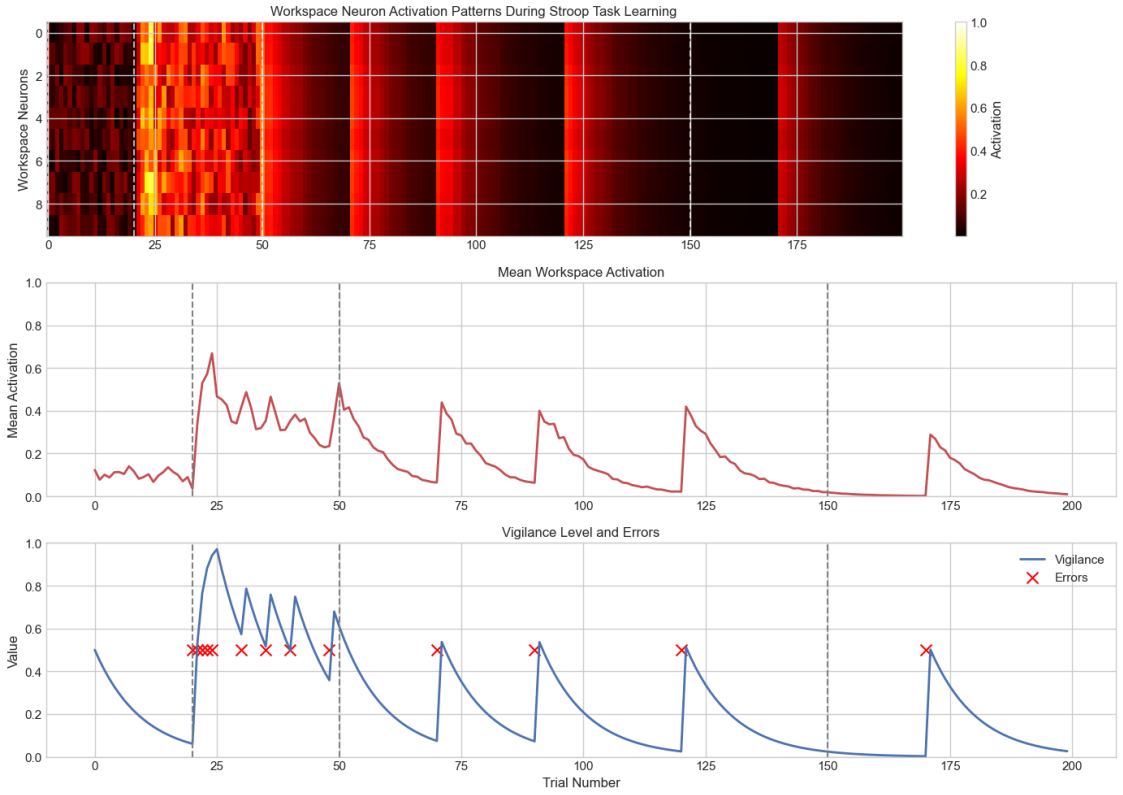
```
    workspace_activity[:, i] *= vigilance[i]

# Phase 2: Effortful execution - stable but high workspace activation
stable_pattern = np.random.uniform(0.7, 0.9, n_workspace_units)
for i in range(50, 150):
    if errors[i] == 1:  # After error, increased activation
        workspace_activity[:, i] = stable_pattern * 1.1
    else:
        workspace_activity[:, i] = stable_pattern * (0.9 + 0.2 * np.random.
 ↪random())

    # Scale by vigilance
    workspace_activity[:, i] *= vigilance[i]

# Phase 3: Progressive routinization - decreasing workspace activation
for i in range(150, n_trials):
    routinization_factor = 1 - 0.5 * (i - 150) / (n_trials - 150)

    if errors[i] == 1:  # After error, temporary return to high activation
        workspace_activity[:, i] = stable_pattern * 1.1
    else:
        workspace_activity[:, i] = (
            stable_pattern * routinization_factor * (0.9 + 0.1 * np.random.
 ↪random())
        )

    # Scale by vigilance
    workspace_activity[:, i] *= vigilance[i]

# Calculate mean workspace activation
for i in range(n_trials):
    mean_workspace[i] = np.mean(workspace_activity[:, i])

# Plotting the workspace dynamics
plt.figure(figsize=(14, 10))

# Visualize workspace neuron activity
plt.subplot(3, 1, 1)
plt.imshow(
    workspace_activity, aspect="auto", cmap="hot", interpolation="nearest",␣
 ↪vmax=1.0
)
plt.colorbar(label="Activation")
plt.title("Workspace Neuron Activation Patterns During Stroop Task Learning")
plt.ylabel("Workspace Neurons")

# Add task phase markings
```

```
phase_names = ["Routine", "Search", "Effortful Execution", "Routinization"]
phase_starts = [0, 20, 50, 150]
for i, (name, start) in enumerate(zip(phase_names, phase_starts)):
    if i < len(phase_starts) - 1:
        end = phase_starts[i + 1]
    else:
        end = n_trials
    plt.annotate(
        name,
        xy=(start + (end - start) / 2, -1),
        xytext=(start + (end - start) / 2, -3),
        arrowprops=dict(arrowstyle="->"),
        ha="center",
    )
    plt.axvline(x=start, color="w", linestyle="--", alpha=0.7)

# Plot mean workspace activity
plt.subplot(3, 1, 2)
plt.plot(mean_workspace, "r-", linewidth=2)
plt.title("Mean Workspace Activation")
plt.ylabel("Mean Activation")
plt.axvline(x=20, color="gray", linestyle="--")
plt.axvline(x=50, color="gray", linestyle="--")
plt.axvline(x=150, color="gray", linestyle="--")
plt.ylim(0, 1)

# Plot vigilance and errors
plt.subplot(3, 1, 3)
plt.plot(vigilance, "b-", linewidth=2, label="Vigilance")
plt.scatter(
    np.where(errors == 1)[0],
    errors[errors == 1] * 0.5,
    color="red",
    marker="x",
    s=100,
    label="Errors",
)
plt.title("Vigilance Level and Errors")
plt.xlabel("Trial Number")
plt.ylabel("Value")
plt.axvline(x=20, color="gray", linestyle="--")
plt.axvline(x=50, color="gray", linestyle="--")
plt.axvline(x=150, color="gray", linestyle="--")
plt.ylim(0, 1)
plt.legend()

plt.tight_layout()
```

```
plt.show()
```

0.2 Simulating Workspace Dynamics in the Stroop Task

The following simulation tracks the workspace activation during different phases of learning the Stroop task. We'll see how workspace activity changes across: 1. Initial routine tasks (no workspace needed) 2. Introduction of the effortful task 3. Search phase (variable workspace activation) 4. Effortful execution phase (stable workspace activation) 5. Error recovery (increased workspace activation) 6. Routinization (declining workspace activation)



[3]:
```
display(
    Markdown(
        """
## Simulating the Attentional Gating in the Stroop Task

The figure below shows the role of workspace neurons in resolving Stroop␣
↪interference.
It demonstrates how descending attentional connections can:
1. Amplify the weaker color pathway
2. Suppress the stronger word pathway
3. Reverse the normal dominance of word reading over color naming
```

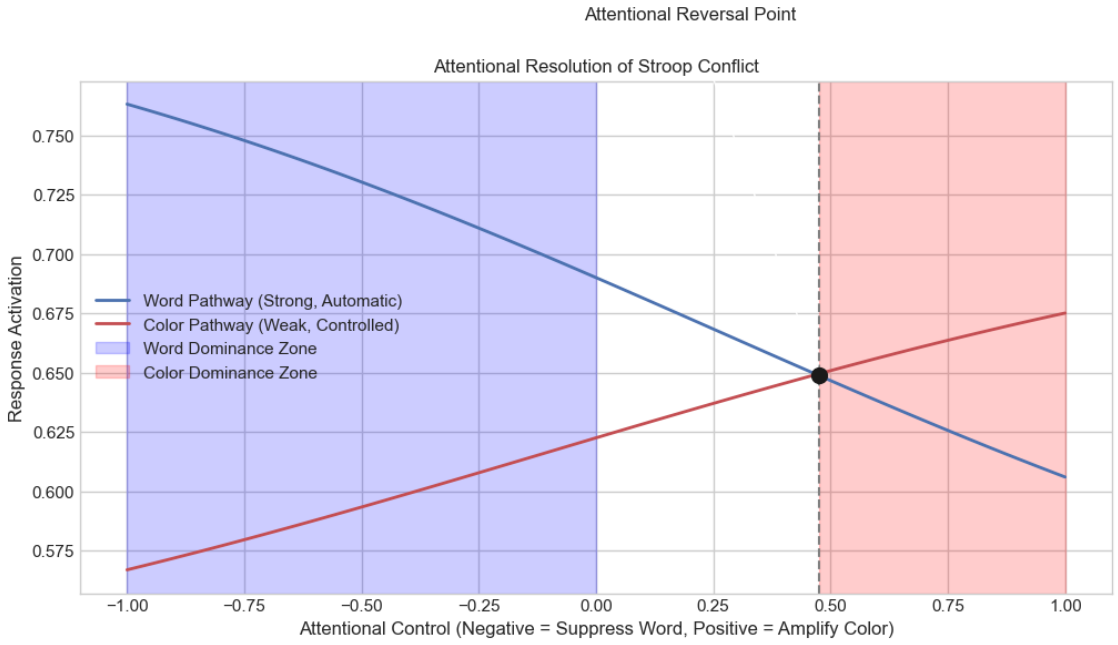
```
This mechanism corresponds to Figure 2 in the paper, which shows how attentional␣
↪amplification reverses
the relation between conflicting word and color inputs.
"""
    )
)

# Simulate the Stroop task conflict resolution through attentional gating
# Parameters for relative strength of word and color pathways
word_strength = 0.8  # Stronger automatic pathway
color_strength = 0.5  # Weaker pathway requiring attention

# Range of attentional modulation values
attention_values = np.linspace(-1, 1, 100)

# Calculate response strengths under different attentional conditions
word_responses = np.zeros(len(attention_values))
color_responses = np.zeros(len(attention_values))

for i, att in enumerate(attention_values):
    # Attentional modulation of word pathway (negative = suppression)
    word_mod = modulating_function(-att)  # Negative attention to suppress word

    # Attentional modulation of color pathway (positive = amplification)
    color_mod = modulating_function(att)  # Positive attention to amplify color

    # Final response strengths
    word_responses[i] = sigmoid(word_strength * word_mod)
    color_responses[i] = sigmoid(color_strength * color_mod)

# Plotting the effect of attention on pathway strengths
plt.figure(figsize=(12, 6))

# Plot response strengths
plt.plot(
    attention_values,
    word_responses,
    "b-",
    linewidth=2,
    label="Word Pathway (Strong, Automatic)",
)
plt.plot(
    attention_values,
    color_responses,
    "r-",
    linewidth=2,
    label="Color Pathway (Weak, Controlled)",
```

```
)

# Mark the crossover point where attention reverses the dominance
crossover_idx = np.argmin(np.abs(word_responses - color_responses))
crossover_att = attention_values[crossover_idx]
crossover_resp = word_responses[crossover_idx]

plt.axvline(x=crossover_att, color="gray", linestyle="--")
plt.scatter([crossover_att], [crossover_resp], color="k", s=100, zorder=5)
plt.annotate(
    "Attentional Reversal Point",
    xy=(crossover_att, crossover_resp),
    xytext=(crossover_att - 0.5, crossover_resp + 0.15),
    arrowprops=dict(arrowstyle="->"),
)

# Mark regions
plt.axvspan(-1, 0, alpha=0.2, color="blue", label="Word Dominance Zone")
plt.axvspan(crossover_att, 1, alpha=0.2, color="red", label="Color Dominance␣
 ↪Zone")

plt.title("Attentional Resolution of Stroop Conflict")
plt.xlabel("Attentional Control (Negative = Suppress Word, Positive = Amplify␣
 ↪Color)")
plt.ylabel("Response Activation")
plt.legend()
plt.grid(True)
plt.show()
```

0.3 Simulating the Attentional Gating in the Stroop Task

The figure below shows the role of workspace neurons in resolving Stroop interference. It demonstrates how descending attentional connections can: 1. Amplify the weaker color pathway 2. Suppress the stronger word pathway 3. Reverse the normal dominance of word reading over color naming

This mechanism corresponds to Figure 2 in the paper, which shows how attentional amplification reverses the relation between conflicting word and color inputs.



[5]:
```
display(
    Markdown(
        """
## Simulating Learning During the Stroop Task

This graph shows how learning (through the reward-modulated Hebbian rule)␣
↪affects the Stroop task performance.
The model learns to strengthen color-to-name connections and weaken word-to-name␣
↪connections during the
effortful task of naming colors in the presence of conflicting words.
"""
    )
)

# Simulate learning in the Stroop task through the Hebbian rule
n_trials = 100

# Initial connection strengths (as in the paper)
word_to_name_weights = np.ones(n_trials) * 5.0  # Strong initial connections
color_to_name_weights = np.ones(n_trials) * 2.0  # Weaker initial connections

# Simulated activities
word_activity = 0.9  # Strong word activation
color_activity = 0.6  # Weaker color activation
name_activity = np.zeros(n_trials)
```

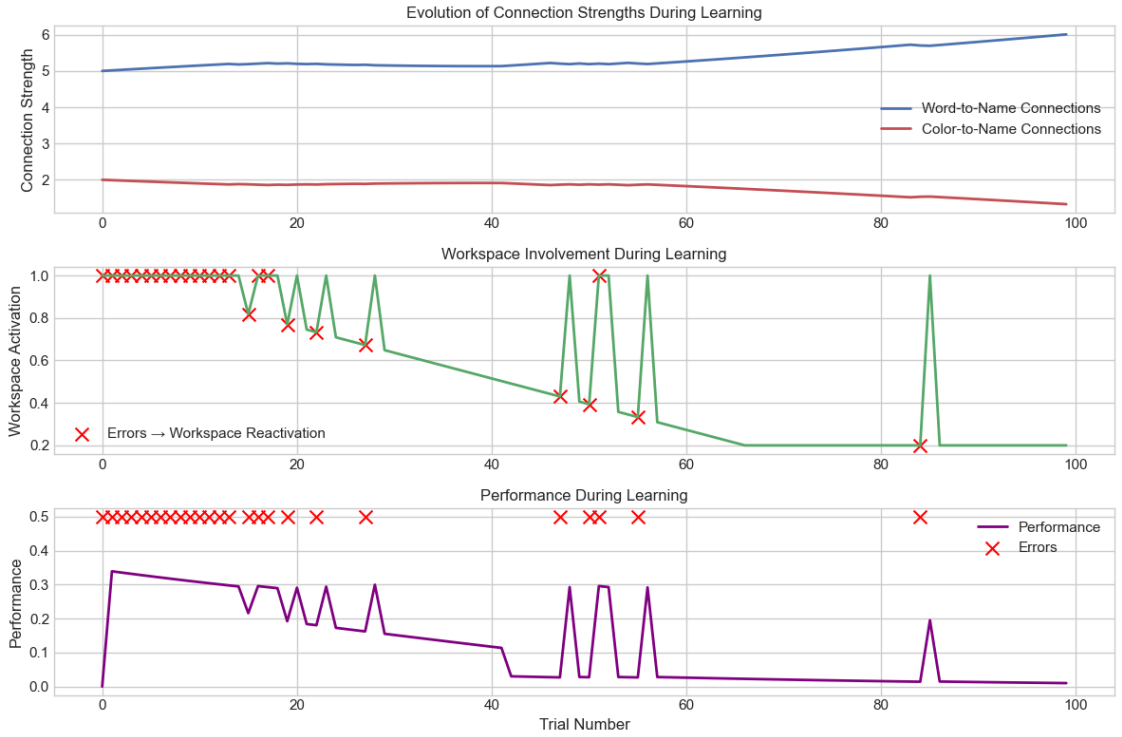
```
# Initialize performance measure
performance = np.zeros(n_trials)
errors = np.zeros(n_trials)
workspace_involvement = np.zeros(n_trials)

# Learning parameters
learning_rate = 0.03
reward = np.ones(n_trials)  # +1 for correct trials

# Set some random errors (more frequent early in training)
error_prob = np.exp(-np.linspace(0, 5, n_trials))
error_trials = np.random.random(n_trials) < error_prob
reward[error_trials] = -1  # -1 reward for incorrect trials
errors[error_trials] = 1

# Simulate workspace involvement (high early, decreasing with learning)
workspace_involvement = 1 - 0.8 * np.minimum(1, np.linspace(0, 1.5, n_trials))
# Reset workspace to high after errors
for i in range(1, n_trials):
    if errors[i - 1] == 1:
        workspace_involvement[i] = 1.0

# Simulation loop
for i in range(1, n_trials):
    # Activation of naming unit based on input weights
    # With workspace activation: attentional amplification of color and␣
 ↪suppression of word
    if workspace_involvement[i] > 0.5:
        # Apply attentional modulation
        color_contribution = (
            color_to_name_weights[i - 1]
            * color_activity
            * modulating_function(workspace_involvement[i])
        )
        word_contribution = (
            word_to_name_weights[i - 1]
            * word_activity
            * modulating_function(-workspace_involvement[i])
        )
    else:
        # No attentional modulation
        color_contribution = color_to_name_weights[i - 1] * color_activity
        word_contribution = word_to_name_weights[i - 1] * word_activity

    # Response based on relative strength of pathways
    name_activity[i] = sigmoid(color_contribution - word_contribution + 2)
```

```
    # Update weights using Hebbian learning rule with reward modulation
    color_to_name_weights[i] = update_weights(
        color_to_name_weights[i - 1],
        color_activity,
        name_activity[i],
        reward[i],
        learning_rate,
    )

    word_to_name_weights[i] = update_weights(
        word_to_name_weights[i - 1],
        word_activity,
        name_activity[i],
        -reward[i],
        learning_rate,
    )

    # Calculate performance (probability of correct response)
    # Higher performance when color response dominates word response
    performance[i] = sigmoid(color_contribution - word_contribution)

# Plotting the learning dynamics
plt.figure(figsize=(12, 8))

# Plot connection strengths
plt.subplot(3, 1, 1)
plt.plot(word_to_name_weights, "b-", linewidth=2, label="Word-to-Name␣
 ↪Connections")
plt.plot(color_to_name_weights, "r-", linewidth=2, label="Color-to-Name␣
 ↪Connections")
plt.title("Evolution of Connection Strengths During Learning")
plt.ylabel("Connection Strength")
plt.legend()
plt.grid(True)

# Plot workspace involvement
plt.subplot(3, 1, 2)
plt.plot(workspace_involvement, "g-", linewidth=2)
plt.scatter(
    np.where(errors == 1)[0],
    workspace_involvement[errors == 1],
    color="red",
    marker="x",
    s=100,
    label="Errors → Workspace Reactivation",
)
```

```
plt.title("Workspace Involvement During Learning")
plt.ylabel("Workspace Activation")
plt.legend()
plt.grid(True)

# Plot performance and errors
plt.subplot(3, 1, 3)
plt.plot(performance, "purple", linewidth=2, label="Performance")
plt.scatter(
    np.where(errors == 1)[0],
    errors[errors == 1] * 0.5,
    color="red",
    marker="x",
    s=100,
    label="Errors",
)
plt.title("Performance During Learning")
plt.xlabel("Trial Number")
plt.ylabel("Performance")
plt.legend()
plt.grid(True)

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

0.4 Simulating Learning During the Stroop Task

This graph shows how learning (through the reward-modulated Hebbian rule) affects the Stroop task performance. The model learns to strengthen color-to-name connections and weaken word-to-name connections during the effortful task of naming colors in the presence of conflicting words.


Evolution of Connection Strengths During Learning
Connection Strength
Word-to-Name Connections
Color-to-Name Connections
Workspace Involvement During Learning
Workspace Activation
Errors → Workspace Reactivation
Performance During Learning
Performance
Performance
Errors
Trial Number

[ ]: