

# Appendix A: Math & Python Mini-Refresher

This appendix provides a concise refresher on key mathematical concepts and Python programming skills that are essential for understanding the neuroAI content in this handbook.

## A.1 Mathematical Foundations

### Linear Algebra Essentials

Linear algebra forms the backbone of modern machine learning and neural data analysis.

#### Vectors and Matrices

```
import numpy as np

# Creating vectors
v = np.array([1, 2, 3])
w = np.array([4, 5, 6])

# Vector operations
dot_product = np.dot(v, w)      # Dot product: 32
norm = np.linalg.norm(v)        # Euclidean norm: 3.74
unit_v = v / np.linalg.norm(v)  # Unit vector

# Creating matrices
A = np.array([[1, 2], [3, 4], [5, 6]]) # 3x2 matrix
B = np.array([[1, 2, 3], [4, 5, 6]])   # 2x3 matrix

# Matrix operations
C = np.dot(B, A) # Matrix multiplication: 2x2 matrix
transpose = A.T  # Transpose: 2x3 matrix
```

# Matrix Operations and Decompositions

```
import numpy as np

# Square matrix for demonstration
M = np.array([[4, 2], [2, 3]])

# Matrix inverse
M_inv = np.linalg.inv(M)
print("M * M^-1 =\n", np.dot(M, M_inv)) # Should be identity matrix

# Matrix determinant
det = np.linalg.det(M) # 8.0

# Singular Value Decomposition (SVD)
U, S, Vt = np.linalg.svd(M)
# Reconstruct original matrix
M_reconstructed = np.dot(U, np.dot(np.diag(S), Vt))

# Eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(M)
print("Eigenvalues:", eigenvalues)
print("Eigenvectors:\n", eigenvectors)

# Principal Component Analysis (simplified)
def simple_pca(X, n_components=2):
    """
    Perform PCA on data matrix X

    Parameters:
    - X: Data matrix (samples x features)
    - n_components: Number of principal components to keep

    Returns:
    - X_reduced: Data projected onto principal components
    - components: Principal components
    """
    # Center the data
    X_centered = X - np.mean(X, axis=0)

    # Compute covariance matrix
    cov_matrix = np.cov(X_centered, rowvar=False)

    # Eigendecomposition
    eigenvalues, eigenvectors = np.linalg.eigh(cov_matrix)

    # Sort eigenvalues and eigenvectors in descending order
    idx = eigenvalues.argsort()[::-1]
    eigenvalues = eigenvalues[idx]
    eigenvectors = eigenvectors[:, idx]

    # Select top n_components
    components = eigenvectors[:, :n_components]
```

```
# Project data onto principal components
X_reduced = np.dot(X_centered, components)

return X_reduced, components
```

## Calculus Fundamentals

Calculus concepts are critical for understanding neural network training and many neuroscience models.

# Derivatives and Gradients

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D

# Define a simple function
def f(x):
    return x**2

# Plot function and its derivative
x = np.linspace(-5, 5, 100)
y = f(x)
dydx = 2*x # Analytical derivative

plt.figure(figsize=(10, 6))
plt.plot(x, y, label='f(x) = x2')
plt.plot(x, dydx, label='f\''(x) = 2x')
plt.grid(True)
plt.legend()
plt.title('Function and its Derivative')
plt.xlabel('x')
plt.ylabel('y')

# Multivariate function and gradient
def g(x, y):
    return x**2 + y**2

# Analytical gradient
def grad_g(x, y):
    return np.array([2*x, 2*y])

# Example of gradient descent
def gradient_descent(grad_func, start, learning_rate, n_iterations):
    """Simple gradient descent implementation"""
    path = [start]
    point = start.copy()

    for _ in range(n_iterations):
        gradient = grad_func(point[0], point[1])
        point = point - learning_rate * gradient
        path.append(point.copy())

    return np.array(path)

# Run gradient descent
start = np.array([4.0, 4.0])
path = gradient_descent(grad_g, start, 0.1, 20)

# Plot the function and gradient descent path
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
```

```
X, Y = np.meshgrid(x, y)
Z = g(X, Y)

fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm, alpha=0.6)
ax.scatter(path[:, 0], path[:, 1], g(path[:, 0], path[:, 1]),
           color='black', s=50, label='Gradient Descent Path')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_title('Gradient Descent Optimization')
```

## The Chain Rule and Backpropagation

The chain rule is the foundation of backpropagation, the algorithm used to train neural networks:

```

def chain_rule_example():
    """
    Demonstrate the chain rule for a simple neural network with:
    - Input x
    - Hidden layer h = sigmoid(w1*x + b1)
    - Output y = w2*h + b2
    - Loss L = (y - target)^2
    """
    def sigmoid(z):
        return 1 / (1 + np.exp(-z))

    def forward_pass(x, w1, b1, w2, b2):
        # Hidden layer
        z1 = w1 * x + b1
        h = sigmoid(z1)

        # Output layer
        y = w2 * h + b2

        return y, h, z1

    def compute_gradients(x, target, w1, b1, w2, b2):
        # Forward pass
        y, h, z1 = forward_pass(x, w1, b1, w2, b2)

        # Compute loss
        loss = (y - target)**2

        # Backpropagation
        # dL/dy = 2(y - target)
        dL_dy = 2 * (y - target)

        # dy/dw2 = h
        dL_dw2 = dL_dy * h

        # dy/db2 = 1
        dL_db2 = dL_dy * 1

        # dy/dh = w2
        dL_dh = dL_dy * w2

        # dh/dz1 = h * (1 - h) [derivative of sigmoid]
        dh_dz1 = h * (1 - h)

        # dz1/dw1 = x
        dL_dw1 = dL_dh * dh_dz1 * x

        # dz1/db1 = 1
        dL_db1 = dL_dh * dh_dz1 * 1

        return loss, dL_dw1, dL_db1, dL_dw2, dL_db2

# Initialize parameters

```

```
x = 2.0
target = 0.5
w1, b1 = 0.1, 0.1
w2, b2 = 0.2, 0.2

# Compute gradients
loss, dL_dw1, dL_db1, dL_dw2, dL_db2 = compute_gradients(x, target, w1, b1, w2, b2)

print(f"Loss: {loss:.4f}")
print(f"Gradients:")
print(f"  dL/dw1: {dL_dw1:.4f}")
print(f"  dL/db1: {dL_db1:.4f}")
print(f"  dL/dw2: {dL_dw2:.4f}")
print(f"  dL/db2: {dL_db2:.4f}")

return loss, dL_dw1, dL_db1, dL_dw2, dL_db2
```

## Probability and Statistics

Probability theory is essential for understanding generative models, variational inference, and neural coding.

```

import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

# Normal distribution
mu, sigma = 0, 1
x = np.linspace(-5, 5, 1000)
pdf = stats.norm.pdf(x, mu, sigma)

plt.figure(figsize=(10, 6))
plt.plot(x, pdf)
plt.fill_between(x, pdf, alpha=0.3)
plt.title('Normal Distribution')
plt.xlabel('x')
plt.ylabel('Probability Density')
plt.grid(True)

# Bayes' theorem example: Disease testing
def bayes_theorem_example():
    """
    Example using Bayes' theorem for medical testing:
    - Prior probability of disease:  $P(D) = 0.01$ 
    - True positive rate:  $P(+|D) = 0.95$ 
    - False positive rate:  $P(+|\neg D) = 0.05$ 
    - What is the probability of disease given positive test?  $P(D|+)$ 
    """
    # Prior probability of disease
    p_disease = 0.01

    # Conditional probabilities
    p_positive_given_disease = 0.95      # True positive rate
    p_positive_given_no_disease = 0.05   # False positive rate

    # Calculate joint probabilities
    p_disease_and_positive = p_disease * p_positive_given_disease
    p_no_disease_and_positive = (1 - p_disease) * p_positive_given_no_disease

    # Calculate marginal probability of positive test
    p_positive = p_disease_and_positive + p_no_disease_and_positive

    # Apply Bayes' theorem
    p_disease_given_positive = p_disease_and_positive / p_positive

    print(f"Probability of disease given positive test: {p_disease_given_positive}")

    return p_disease_given_positive

# Information theory – Entropy and KL divergence
def information_theory_example():
    """
    Calculate entropy and KL divergence for discrete distributions
    """
    # Two probability distributions

```



```
p = np.array([0.2, 0.5, 0.3])
q = np.array([0.1, 0.4, 0.5])

# Entropy of p
entropy_p = -np.sum(p * np.log2(p))

# KL divergence between p and q
kl_div = np.sum(p * np.log2(p / q))

print(f"Entropy of p: {entropy_p:.4f} bits")
print(f"KL divergence from p to q: {kl_div:.4f} bits")

return entropy_p, kl_div
```

## A.2 Python Fundamentals

### Core Python

Python's simplicity and readability make it ideal for neuroscience and AI research.

```

# Data structures
# Lists
my_list = [1, 2, 3, 'a', 'b']
my_list.append(4)
print(my_list[0]) # Indexing: 1
print(my_list[-1]) # Negative indexing: 4
print(my_list[1:3]) # Slicing: [2, 3]

# Dictionaries
my_dict = {'name': 'Neuron', 'type': 'Pyramidal', 'location': 'V1'}
print(my_dict['type']) # Accessing value: Pyramidal
my_dict['active'] = True # Adding new key-value pair

# List comprehension
squares = [x**2 for x in range(10)]
even_squares = [x**2 for x in range(10) if x % 2 == 0]

# Functions and lambda expressions
def calculate_firing_rate(spike_count, time_window):
    """Calculate firing rate in Hz"""
    return spike_count / time_window

# Lambda function for the same calculation
firing_rate = lambda spike_count, time_window: spike_count / time_window

# Classes and OOP
class Neuron:
    """Simple neuron model"""

    def __init__(self, rest_potential=-70, threshold=-55):
        self.membrane_potential = rest_potential
        self.rest_potential = rest_potential
        self.threshold = threshold
        self.spike_history = []

    def receive_input(self, input_current):
        """Process input current and update membrane potential"""
        self.membrane_potential += input_current
        if self.membrane_potential >= self.threshold:
            self.spike()
            return True
        return False

    def spike(self):
        """Generate an action potential"""
        self.spike_history.append(1)
        self.membrane_potential = self.rest_potential

    def reset(self):
        """Reset neuron state"""
        self.membrane_potential = self.rest_potential
        self.spike_history = []

```

```
# Creating and using a neuron instance
my_neuron = Neuron()
my_neuron.receive_input(20)
print(f"Membrane potential: {my_neuron.membrane_potential} mV")
```

## Scientific Python Ecosystem

The scientific Python ecosystem provides powerful tools for data analysis, visualization, and modeling.

### NumPy: Numerical Computing

```
import numpy as np

# Creating arrays
a = np.array([1, 2, 3, 4, 5])
b = np.arange(10) # [0, 1, 2, ..., 9]
c = np.linspace(0, 1, 5) # [0.0, 0.25, 0.5, 0.75, 1.0]
zeros = np.zeros((3, 3)) # 3x3 matrix of zeros
ones = np.ones((2, 2)) # 2x2 matrix of ones
rand = np.random.rand(2, 2) # 2x2 matrix of random numbers from U(0,1)

# Array operations
a + 2 # Element-wise addition: [3, 4, 5, 6, 7]
a * 2 # Element-wise multiplication: [2, 4, 6, 8, 10]
a * a # Element-wise product: [1, 4, 9, 16, 25]

# Matrix operations
M = np.array([[1, 2], [3, 4]])
v = np.array([1, 2])
np.dot(M, v) # Matrix-vector product: [5, 11]
eigvals, eigvecs = np.linalg.eig(M) # Eigendecomposition

# Statistical operations
np.mean(a) # Mean: 3.0
np.std(a) # Standard deviation: ~1.41
np.max(a) # Maximum: 5
np.min(a) # Minimum: 1
```

# Matplotlib: Visualization

```
import matplotlib.pyplot as plt
import numpy as np

# Basic line plot
x = np.linspace(0, 10, 100)
y = np.sin(x)
plt.figure(figsize=(10, 6))
plt.plot(x, y, 'b-', label='sin(x)')
plt.title('Sine Function')
plt.xlabel('x')
plt.ylabel('sin(x)')
plt.grid(True)
plt.legend()
plt.show()

# Scatter plot with coloring
n = 100
x = np.random.rand(n)
y = np.random.rand(n)
colors = np.random.rand(n)
sizes = 1000 * np.random.rand(n)

plt.figure(figsize=(10, 6))
plt.scatter(x, y, c=colors, s=sizes, alpha=0.5)
plt.title('Scatter Plot with Size and Color Mapping')
plt.grid(True)
plt.colorbar(label='Color Value')
plt.show()

# Multiple plots in a grid
fig, axs = plt.subplots(2, 2, figsize=(10, 8))

# First plot: line
axs[0, 0].plot(x, y)
axs[0, 0].set_title('Line Plot')

# Second plot: scatter
axs[0, 1].scatter(x, y, c=colors)
axs[0, 1].set_title('Scatter Plot')

# Third plot: histogram
axs[1, 0].hist(y, bins=20)
axs[1, 0].set_title('Histogram')

# Fourth plot: bar chart
axs[1, 1].bar(range(5), np.random.rand(5))
axs[1, 1].set_title('Bar Chart')

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

# Pandas: Data Manipulation

```
import pandas as pd
import numpy as np

# Creating a DataFrame
data = {
    'neuron_id': np.arange(1, 6),
    'type': ['Pyramidal', 'Stellate', 'Pyramidal', 'Basket', 'Pyramidal'],
    'resting_potential': [-70, -65, -72, -68, -71],
    'firing_rate': [5.2, 10.1, 3.5, 15.0, 4.8],
    'location': ['V1', 'V2', 'V1', 'V2', 'MT']
}

df = pd.DataFrame(data)
print(df.head())

# Basic operations
print(df.describe()) # Summary statistics
print(df['firing_rate'].mean()) # Mean of a column

# Filtering
pyramidal_neurons = df[df['type'] == 'Pyramidal']
high_firing = df[df['firing_rate'] > 10]

# Grouping and aggregation
type_stats = df.groupby('type').agg({
    'resting_potential': 'mean',
    'firing_rate': ['mean', 'std', 'count']
})
print(type_stats)

# Adding a new column
df['active'] = df['firing_rate'] > 5
```

# SciPy: Scientific Computing

```
from scipy import stats, optimize, integrate, signal
import numpy as np
import matplotlib.pyplot as plt

# Statistical tests
x = np.random.normal(0, 1, 100)
y = np.random.normal(2, 1, 100)
t_stat, p_value = stats.ttest_ind(x, y)
print(f"t-statistic: {t_stat:.4f}, p-value: {p_value:.4f}")

# Optimization
def objective(x):
    return x[0]**2 + x[1]**2

result = optimize.minimize(objective, [1, 1])
print(f"Minimum found at: {result.x}")
print(f"Minimum value: {result.fun}")

# Integration
def f(x):
    return x**2

integral, error = integrate.quad(f, 0, 1)
print(f"Integral of x^2 from 0 to 1: {integral:.6f} ± {error:.6f}")

# Signal processing: filtering
t = np.linspace(0, 1, 1000, endpoint=False)
noise = np.random.normal(0, 0.1, t.shape)
signal_clean = np.sin(2 * np.pi * 10 * t) # 10 Hz sine wave
signal_noisy = signal_clean + noise

# Design a low-pass filter
b, a = signal.butter(4, 0.2) # 4th order Butterworth filter with cutoff at 0.2 *
filtered_signal = signal.filtfilt(b, a, signal_noisy)

plt.figure(figsize=(10, 6))
plt.plot(t, signal_noisy, 'gray', alpha=0.6, label='Noisy signal')
plt.plot(t, signal_clean, 'g--', linewidth=2, label='Original signal')
plt.plot(t, filtered_signal, 'b-', linewidth=1.5, label='Filtered signal')
plt.legend()
plt.xlabel('Time [s]')
plt.title('Signal Filtering Example')
plt.grid(True)
plt.show()
```

# Machine Learning Libraries

## PyTorch Fundamentals

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt

# Create a simple neural network
class SimpleNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleNN, self).__init__()
        self.layer1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.layer2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.layer1(x)
        x = self.relu(x)
        x = self.layer2(x)
        return x

# Create a synthetic dataset
def generate_data(n_samples=100):
    # Generate two-class classification data
    np.random.seed(42)
    X = np.random.randn(n_samples, 2)
    y = (X[:, 0] + X[:, 1] > 0).astype(np.int64)

    # Convert to PyTorch tensors
    X_tensor = torch.FloatTensor(X)
    y_tensor = torch.LongTensor(y)

    return X_tensor, y_tensor

# Training function
def train_model(model, X, y, epochs=1000, lr=0.01):
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=lr)

    # Track losses
    losses = []

    for epoch in range(epochs):
        # Forward pass
        outputs = model(X)
        loss = criterion(outputs, y)
```

```

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

        # Save loss
        losses.append(loss.item())

        # Print progress
        if (epoch+1) % 100 == 0:
            print(f'Epoch {epoch+1}/{epochs}, Loss: {loss.item():.4f}')

    return losses

# Prepare data
X, y = generate_data(100)

# Create and train model
model = SimpleNN(input_size=2, hidden_size=10, output_size=2)
losses = train_model(model, X, y)

# Plot decision boundary
def plot_decision_boundary(model, X, y):
    # Define grid for plotting
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
                          np.arange(y_min, y_max, 0.01))

    # Get predictions for all grid points
    grid = torch.FloatTensor(np.c_[xx.ravel(), yy.ravel()])
    with torch.no_grad():
        outputs = model(grid)
        _, predictions = torch.max(outputs, 1)

    # Reshape predictions back to grid
    predictions = predictions.reshape(xx.shape)

    # Plot decision boundary
    plt.figure(figsize=(10, 6))
    plt.contourf(xx, yy, predictions, alpha=0.3)

    # Plot training points
    scatter = plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', s=50)
    plt.legend(*scatter.legend_elements(), title="Classes")

    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.title('Decision Boundary')
    plt.grid(True)
    plt.show()

# Plot results
plt.figure(figsize=(10, 6))

```



```
plt.plot(losses)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss')
plt.grid(True)
plt.show()

plot_decision_boundary(model, X, y)
```

# TensorFlow/Keras Example

```
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt

# Load a built-in dataset (MNIST)
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Preprocess data
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Build a convolutional neural network
model = keras.Sequential([
    keras.layers.Input(shape=(28, 28)),
    keras.layers.Reshape((28, 28, 1)),
    keras.layers.Conv2D(32, kernel_size=(3, 3), activation='relu'),
    keras.layers.MaxPooling2D(pool_size=(2, 2)),
    keras.layers.Conv2D(64, kernel_size=(3, 3), activation='relu'),
    keras.layers.MaxPooling2D(pool_size=(2, 2)),
    keras.layers.Flatten(),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(10, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train the model (using a subset for demonstration)
history = model.fit(x_train[:10000], y_train[:10000],
                    validation_split=0.2,
                    batch_size=128,
                    epochs=5)

# Evaluate on test data
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f'Test accuracy: {test_acc:.4f}')

# Plot training history
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
```

```

plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.tight_layout()
plt.show()

# Visualize predictions
def plot_predictions(model, x_test, y_test, n=5):
    # Get model predictions
    predictions = model.predict(x_test[:n])
    predicted_classes = np.argmax(predictions, axis=1)

    plt.figure(figsize=(12, 4))
    for i in range(n):
        plt.subplot(1, n, i+1)
        plt.imshow(x_test[i], cmap='gray')
        color = 'green' if predicted_classes[i] == y_test[i] else 'red'
        plt.title(f"Pred: {predicted_classes[i]}\nTrue: {y_test[i]}", color=color)
        plt.axis('off')
    plt.tight_layout()
    plt.show()

plot_predictions(model, x_test, y_test)

```

## A.3 Neuroscience Data Processing

### Common Data Formats

Neuroscience data comes in various formats, each with its own characteristics:

# Spike Trains and Rasters

```
import numpy as np
import matplotlib.pyplot as plt

# Simulate spike trains for 5 neurons over 1 second
def simulate_poisson_spike_train(rate, t_max, dt):
    """Simulate a Poisson spike train"""
    t = np.arange(0, t_max, dt)
    n_steps = len(t)

    # Probability of spike in each time bin
    prob_spike = rate * dt

    # Generate spikes
    spikes = np.random.rand(n_steps) < prob_spike

    # Get spike times
    spike_times = t[spikes]

    return spike_times, spikes

# Simulation parameters
t_max = 1.0 # seconds
dt = 0.001 # 1 ms resolution
n_neurons = 5
firing_rates = [5, 10, 15, 20, 25] # Hz

# Generate spike trains
spike_trains = []
spike_rasters = []

for rate in firing_rates:
    spike_times, spikes = simulate_poisson_spike_train(rate, t_max, dt)
    spike_trains.append(spike_times)
    spike_rasters.append(spikes)

# Plot raster plot
plt.figure(figsize=(12, 6))

# Plot spike raster
plt.subplot(2, 1, 1)
for i, (rate, spike_times) in enumerate(zip(firing_rates, spike_trains)):
    plt.plot(spike_times, np.ones_like(spike_times) * i, '|', markersize=10)
plt.yticks(range(n_neurons), [f'{rate} Hz' for rate in firing_rates])
plt.xlabel('Time (s)')
plt.ylabel('Neuron')
plt.title('Spike Raster Plot')

# Plot PSTH (Peri-Stimulus Time Histogram)
plt.subplot(2, 1, 2)
bin_size = 0.05 # 50 ms bins
bins = np.arange(0, t_max + bin_size, bin_size)
```

```
t = np.arange(0, t_max, dt)

for i, spikes in enumerate(spike_rasters):
    counts, _ = np.histogram(t[spikes], bins=bins)
    rate = counts / bin_size # Convert to Hz
    plt.step(bins[:-1], rate, label=f'Neuron {i+1}')

plt.xlabel('Time (s)')
plt.ylabel('Firing Rate (Hz)')
plt.title('Peri-Stimulus Time Histogram (PSTH)')
plt.legend()
plt.tight_layout()
plt.show()
```

# Time Series Data

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal

# Simulate EEG data (multi-channel time series)
def simulate_eeg(fs=250, duration=5, n_channels=4):
    """
    Simulate multi-channel EEG data with different frequency bands

    Parameters:
    - fs: Sampling frequency (Hz)
    - duration: Duration of the signal (seconds)
    - n_channels: Number of channels

    Returns:
    - time: Time points
    - eeg: Simulated EEG data (channels x time points)
    """
    t = np.arange(0, duration, 1/fs)
    n_points = len(t)

    # Generate different frequency components
    delta = np.sin(2 * np.pi * 2 * t) # 2 Hz (delta band: 0.5-4 Hz)
    theta = 0.5 * np.sin(2 * np.pi * 6 * t) # 6 Hz (theta band: 4-8 Hz)
    alpha = 0.3 * np.sin(2 * np.pi * 10 * t) # 10 Hz (alpha band: 8-13 Hz)
    beta = 0.2 * np.sin(2 * np.pi * 20 * t) # 20 Hz (beta band: 13-30 Hz)
    gamma = 0.1 * np.sin(2 * np.pi * 40 * t) # 40 Hz (gamma band: >30 Hz)

    # Create channels with different mixtures of frequency components
    eeg = np.zeros((n_channels, n_points))

    # Channel 1: Mostly delta and theta
    eeg[0] = delta + 0.5*theta + 0.2*alpha + 0.1*beta + 0.05*gamma + 0.2*np.random.randn(n_points)

    # Channel 2: Mostly alpha
    eeg[1] = 0.2*delta + 0.3*theta + alpha + 0.2*beta + 0.1*gamma + 0.2*np.random.randn(n_points)

    # Channel 3: Mostly beta
    eeg[2] = 0.1*delta + 0.2*theta + 0.3*alpha + beta + 0.3*gamma + 0.2*np.random.randn(n_points)

    # Channel 4: Mostly gamma
    eeg[3] = 0.05*delta + 0.1*theta + 0.2*alpha + 0.3*beta + gamma + 0.2*np.random.randn(n_points)

    return t, eeg

# Simulate EEG data
fs = 250 # sampling frequency (Hz)
t, eeg = simulate_eeg(fs=fs, duration=5, n_channels=4)

# Plot time domain signals
plt.figure(figsize=(12, 8))
```

```

channel_names = ['Frontal (Fz)', 'Central (Cz)', 'Parietal (Pz)', 'Occipital (Oz)']

for i in range(4):
    plt.subplot(4, 1, i+1)
    plt.plot(t, eeg[i])
    plt.ylabel(channel_names[i])
    if i == 0:
        plt.title('Simulated EEG Data')
    if i == 3:
        plt.xlabel('Time (s)')
plt.tight_layout()
plt.show()

# Compute and plot power spectrum
plt.figure(figsize=(12, 6))

for i in range(4):
    plt.subplot(2, 2, i+1)

    # Compute power spectrum
    f, Pxx = signal.welch(eeg[i], fs=fs, nperseg=fs)

    # Plot power spectrum
    plt.semilogy(f, Pxx)
    plt.xlabel('Frequency (Hz)')
    plt.ylabel('Power Spectral Density')
    plt.title(f'Channel: {channel_names[i]}')
    plt.grid(True)
    plt.axvline(x=4, color='r', linestyle='--', alpha=0.3)
    plt.axvline(x=8, color='r', linestyle='--', alpha=0.3)
    plt.axvline(x=13, color='r', linestyle='--', alpha=0.3)
    plt.axvline(x=30, color='r', linestyle='--', alpha=0.3)
    plt.xticks([0, 4, 8, 13, 30, 50])
    plt.xlim(0, 50)

plt.tight_layout()
plt.show()

```

## Preprocessing Techniques

Preprocessing is a critical step in neuroscience data analysis:

```

import numpy as np
import matplotlib.pyplot as plt
from scipy import signal

# Generate noisy signal
def generate_noisy_eeg(fs=250, duration=5):
    """Generate a noisy EEG-like signal with various artifacts"""
    t = np.arange(0, duration, 1/fs)
    n_points = len(t)

    # Generate clean signal (mixture of oscillations)
    clean = (0.5 * np.sin(2 * np.pi * 10 * t) + # Alpha (10 Hz)
             0.25 * np.sin(2 * np.pi * 5 * t) + # Theta (5 Hz)
             0.1 * np.sin(2 * np.pi * 20 * t)) # Beta (20 Hz)

    # Add noise
    noise = 0.2 * np.random.randn(n_points)

    # Add power line noise (50 Hz)
    line_noise = 0.15 * np.sin(2 * np.pi * 50 * t)

    # Add ocular artifact (slow wave at specific timepoints)
    artifact = np.zeros(n_points)
    artifact_times = [1.2, 3.7] # Times of artifacts (seconds)

    for art_time in artifact_times:
        idx = int(art_time * fs)
        # Create a slow wave artifact
        window = signal.gaussian(100, std=20)
        if idx + len(window) <= n_points:
            artifact[idx:idx+len(window)] += 2 * window

    # Combine all components
    noisy_signal = clean + noise + line_noise + artifact

    return t, clean, noisy_signal, artifact

# Generate signal
fs = 250
t, clean, noisy, artifact = generate_noisy_eeg(fs=fs)

# Plot raw signals
plt.figure(figsize=(12, 8))
plt.subplot(3, 1, 1)
plt.plot(t, clean)
plt.title('Clean EEG Signal')
plt.ylabel('Amplitude')

plt.subplot(3, 1, 2)
plt.plot(t, noisy)
plt.title('Noisy EEG Signal with Artifacts')
plt.ylabel('Amplitude')

```



```

plt.subplot(3, 1, 3)
plt.plot(t, artifact)
plt.title('Artifacts Component')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')

plt.tight_layout()
plt.show()

# Apply filtering
def apply_filters(signal_data, fs):
    """Apply various filters to the signal"""
    # Notch filter for line noise (50 Hz)
    b_notch, a_notch = signal.iirnotch(50, 30, fs)
    notch_filtered = signal.filtfilt(b_notch, a_notch, signal_data)

    # Bandpass filter for EEG frequencies of interest (1-40 Hz)
    b_bandpass, a_bandpass = signal.butter(4, [1, 40], fs=fs, btype='bandpass')
    bandpass_filtered = signal.filtfilt(b_bandpass, a_bandpass, notch_filtered)

    return notch_filtered, bandpass_filtered

# Apply filters
notch_filtered, bandpass_filtered = apply_filters(noisy, fs)

# Plot filtered signals
plt.figure(figsize=(12, 10))
plt.subplot(4, 1, 1)
plt.plot(t, clean)
plt.title('Original Clean EEG Signal')
plt.ylabel('Amplitude')

plt.subplot(4, 1, 2)
plt.plot(t, noisy)
plt.title('Noisy EEG Signal with Artifacts')
plt.ylabel('Amplitude')

plt.subplot(4, 1, 3)
plt.plot(t, notch_filtered)
plt.title('After Notch Filter (50 Hz removed)')
plt.ylabel('Amplitude')

plt.subplot(4, 1, 4)
plt.plot(t, bandpass_filtered)
plt.title('After Bandpass Filter (1-40 Hz)')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')

plt.tight_layout()
plt.show()

# Compute and plot spectrograms
plt.figure(figsize=(12, 10))

```

```

plt.subplot(3, 1, 1)
f, t_spec, Sxx = signal.spectrogram(noisy, fs=fs, nperseg=fs//2, noverlap=fs//4)
plt.pcolormesh(t_spec, f, 10 * np.log10(Sxx), shading='gouraud')
plt.ylabel('Frequency [Hz]')
plt.title('Spectrogram of Noisy Signal')
plt.colorbar(label='PSD [dB]')
plt.ylim(0, 60)

plt.subplot(3, 1, 2)
f, t_spec, Sxx = signal.spectrogram(notch_filtered, fs=fs, nperseg=fs//2, noverlap=fs//4)
plt.pcolormesh(t_spec, f, 10 * np.log10(Sxx), shading='gouraud')
plt.ylabel('Frequency [Hz]')
plt.title('Spectrogram after Notch Filter')
plt.colorbar(label='PSD [dB]')
plt.ylim(0, 60)

plt.subplot(3, 1, 3)
f, t_spec, Sxx = signal.spectrogram(bandpass_filtered, fs=fs, nperseg=fs//2, noverlap=fs//4)
plt.pcolormesh(t_spec, f, 10 * np.log10(Sxx), shading='gouraud')
plt.xlabel('Time [s]')
plt.ylabel('Frequency [Hz]')
plt.title('Spectrogram after Bandpass Filter')
plt.colorbar(label='PSD [dB]')
plt.ylim(0, 60)

plt.tight_layout()
plt.show()

```

## A.4 Code Examples

Here are some integrated examples demonstrating common tasks in computational neuroscience and AI:

# Neural Decoding Example

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, confusion_matrix
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset

# Simulate neural data and behavior
def simulate_neural_decoding_data(n_neurons=50, n_samples=1000, n_classes=3):
    """
    Simulate neural population activity encoding different behaviors

    Parameters:
    - n_neurons: Number of neurons
    - n_samples: Number of samples (trials)
    - n_classes: Number of behavioral classes

    Returns:
    - X: Neural activity data (n_samples, n_neurons)
    - y: Behavioral labels (n_samples,)
    """
    np.random.seed(42)

    # Create class-specific activity patterns
    neuron_templates = np.random.randn(n_classes, n_neurons) * 2

    # Initialize data
    X = np.zeros((n_samples, n_neurons))
    y = np.zeros(n_samples, dtype=int)

    # Generate samples
    samples_per_class = n_samples // n_classes
    for c in range(n_classes):
        start_idx = c * samples_per_class
        end_idx = (c + 1) * samples_per_class if c < n_classes - 1 else n_samples

        # Assign class labels
        y[start_idx:end_idx] = c

        # Generate neural activity based on class template
        for i in range(start_idx, end_idx):
            # Add noise to template
            X[i] = neuron_templates[c] + np.random.randn(n_neurons) * 0.5

    return X, y

# Generate simulated data
X, y = simulate_neural_decoding_data(n_neurons=100, n_samples=1000, n_classes=4)
```

```

# Preprocess data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_s

# Standardize features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Convert to PyTorch tensors
X_train_tensor = torch.FloatTensor(X_train)
y_train_tensor = torch.LongTensor(y_train)
X_test_tensor = torch.FloatTensor(X_test)
y_test_tensor = torch.LongTensor(y_test)

# Create dataset and dataloader
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)

# Define neural network model
class NeuralDecoder(nn.Module):
    def __init__(self, n_neurons, n_hidden, n_classes):
        super(NeuralDecoder, self).__init__()
        self.fc1 = nn.Linear(n_neurons, n_hidden)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(n_hidden, n_classes)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.dropout(x)
        x = self.fc2(x)
        return x

# Initialize model and optimizer
n_neurons = X_train.shape[1]
n_hidden = 64
n_classes = len(np.unique(y))
model = NeuralDecoder(n_neurons, n_hidden, n_classes)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Train the model
n_epochs = 20
train_losses = []

for epoch in range(n_epochs):
    model.train()
    epoch_loss = 0

    for inputs, labels in train_loader:
        # Forward pass
        outputs = model(inputs)

```

```

        loss = criterion(outputs, labels)

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

        epoch_loss += loss.item()

    avg_loss = epoch_loss / len(train_loader)
    train_losses.append(avg_loss)
    print(f'Epoch {epoch+1}/{n_epochs}, Loss: {avg_loss:.4f}')

# Evaluate model
model.eval()
with torch.no_grad():
    y_pred = model(X_test_tensor)
    _, predicted = torch.max(y_pred, 1)

accuracy = accuracy_score(y_test, predicted.numpy())
conf_matrix = confusion_matrix(y_test, predicted.numpy())

print(f'Test Accuracy: {accuracy:.4f}')

# Visualize results
plt.figure(figsize=(15, 5))

# Plot training loss
plt.subplot(1, 2, 1)
plt.plot(train_losses)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss')
plt.grid(True)

# Plot confusion matrix
plt.subplot(1, 2, 2)
plt.imshow(conf_matrix, cmap='Blues')
plt.colorbar()
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.xticks(np.arange(n_classes))
plt.yticks(np.arange(n_classes))

# Add text annotations to confusion matrix
thresh = conf_matrix.max() / 2
for i in range(conf_matrix.shape[0]):
    for j in range(conf_matrix.shape[1]):
        plt.text(j, i, f'{conf_matrix[i, j]}',
                 ha="center", va="center",
                 color="white" if conf_matrix[i, j] > thresh else "black")

```

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

# Simple Neuron Simulation

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

# Hodgkin-Huxley Neuron Model
class HodgkinHuxleyNeuron:
    """
    Implementation of the Hodgkin-Huxley neuron model

    The model describes the evolution of membrane voltage V and
    three gating variables m, n, and h using a system of ODEs
    """

    def __init__(self):
        # Maximal conductances (mS/cm^2)
        self.g_Na = 120.0 # Sodium
        self.g_K = 36.0 # Potassium
        self.g_L = 0.3 # Leak

        # Reversal potentials (mV)
        self.E_Na = 50.0 # Sodium
        self.E_K = -77.0 # Potassium
        self.E_L = -54.387 # Leak

        # Membrane capacitance (μF/cm^2)
        self.C_m = 1.0

    def alpha_m(self, V):
        """Na+ activation rate"""
        return 0.1 * (V + 40.0) / (1.0 - np.exp(-(V + 40.0) / 10.0))

    def beta_m(self, V):
        """Na+ deactivation rate"""
        return 4.0 * np.exp(-(V + 65.0) / 18.0)

    def alpha_h(self, V):
        """Na+ inactivation rate"""
        return 0.07 * np.exp(-(V + 65.0) / 20.0)

    def beta_h(self, V):
        """Na+ deinactivation rate"""
        return 1.0 / (1.0 + np.exp(-(V + 35.0) / 10.0))

    def alpha_n(self, V):
        """K+ activation rate"""
        return 0.01 * (V + 55.0) / (1.0 - np.exp(-(V + 55.0) / 10.0))

    def beta_n(self, V):
        """K+ deactivation rate"""
        return 0.125 * np.exp(-(V + 65) / 80.0)
```

```

def I_Na(self, V, m, h):
    """Na+ current"""
    return self.g_Na * m**3 * h * (V - self.E_Na)

def I_K(self, V, n):
    """K+ current"""
    return self.g_K * n**4 * (V - self.E_K)

def I_L(self, V):
    """Leak current"""
    return self.g_L * (V - self.E_L)

def dALLdt(self, X, t, I_ext):
    """
    Compute derivatives for the complete system of ODEs

    Parameters:
    - X: State vector [V, m, h, n]
    - t: Time
    - I_ext: External current

    Returns:
    - dXdt: Derivatives [dV/dt, dm/dt, dh/dt, dn/dt]
    """
    V, m, h, n = X

    # Steady-state values
    m_inf = self.alpha_m(V) / (self.alpha_m(V) + self.beta_m(V))
    h_inf = self.alpha_h(V) / (self.alpha_h(V) + self.beta_h(V))
    n_inf = self.alpha_n(V) / (self.alpha_n(V) + self.beta_n(V))

    # Time constants
    tau_m = 1.0 / (self.alpha_m(V) + self.beta_m(V))
    tau_h = 1.0 / (self.alpha_h(V) + self.beta_h(V))
    tau_n = 1.0 / (self.alpha_n(V) + self.beta_n(V))

    # Ionic currents
    I_Na = self.I_Na(V, m, h)
    I_K = self.I_K(V, n)
    I_L = self.I_L(V)

    # Membrane voltage derivative
    dVdt = (I_ext - I_Na - I_K - I_L) / self.C_m

    # Gating variables derivatives
    dmdt = (m_inf - m) / tau_m
    dhdt = (h_inf - h) / tau_h
    dndt = (n_inf - n) / tau_n

    return [dVdt, dmdt, dhdt, dndt]

def simulate(self, t, I_ext_func, V0=-65.0):
    """
    Simulate the neuron

```



```
Parameters:
- t: Time array
- I_ext_func: Function that returns external current at time t
- V0: Initial membrane voltage
```

```
Returns:
```

```
- result: Solution array with columns [V, m, h, n]
```

```
"""
# Initial conditions [V, m, h, n]
m0 = self.alpha_m(V0) / (self.alpha_m(V0) + self.beta_m(V0))
h0 = self.alpha_h(V0) / (self.alpha_h(V0) + self.beta_h(V0))
n0 = self.alpha_n(V0) / (self.alpha_n(V0) + self.beta_n(V0))
X0 = [V0, m0, h0, n0]

# Create I_ext array
I_ext = np.array([I_ext_func(time) for time in t])

# Solve ODE system for each time step
result = np.zeros((len(t), 4))
result[0] = X0

for i in range(1, len(t)):
    t_span = [t[i-1], t[i]]
    sol = odeint(self.dALLdt, result[i-1], t_span, args=(I_ext[i],))
    result[i] = sol[-1]

return result
```

```
# Create and simulate the Hodgkin-Huxley neuron
neuron = HodgkinHuxleyNeuron()
```

```
# Simulation parameters
t_max = 50.0 # ms
dt = 0.01 # ms
t = np.arange(0, t_max, dt)
```

```
# External current function (10μA/cm² pulse from 5ms to 30ms)
```

```
def I_ext(t):
    if 5 <= t <= 30:
        return 10.0
    else:
        return 0.0
```

```
# Run simulation
result = neuron.simulate(t, I_ext)
V, m, h, n = result.T
```

```
# Create I_ext array for plotting
I_ext_array = np.array([I_ext(time) for time in t])
```

```
# Plot results
plt.figure(figsize=(12, 10))
```

```

# Membrane voltage
plt.subplot(4, 1, 1)
plt.plot(t, V)
plt.ylabel('Voltage (mV)')
plt.title('Hodgkin-Huxley Neuron Model')

# Gating variables
plt.subplot(4, 1, 2)
plt.plot(t, m, 'r', label='m (Na+ activation)')
plt.plot(t, h, 'g', label='h (Na+ inactivation)')
plt.plot(t, n, 'b', label='n (K+ activation)')
plt.ylabel('Gating Value')
plt.legend()

# Ionic currents
I_Na = np.array([neuron.I_Na(V[i], m[i], h[i]) for i in range(len(t))])
I_K = np.array([neuron.I_K(V[i], n[i]) for i in range(len(t))])
I_L = np.array([neuron.I_L(V[i]) for i in range(len(t))])

plt.subplot(4, 1, 3)
plt.plot(t, I_Na, 'r', label='Na+ Current')
plt.plot(t, I_K, 'b', label='K+ Current')
plt.plot(t, I_L, 'g', label='Leak Current')
plt.ylabel('Current ( $\mu\text{A}/\text{cm}^2$ )')
plt.legend()

# External current
plt.subplot(4, 1, 4)
plt.plot(t, I_ext_array, 'k')
plt.xlabel('Time (ms)')
plt.ylabel('Stimulus ( $\mu\text{A}/\text{cm}^2$ )')

plt.tight_layout()
plt.show()

```

These examples and refreshers should provide the necessary background for understanding the more advanced concepts presented in the handbook chapters.