Web ブラウザにおける機械学習の実践的導入

—ml5.js を用いた段階的学習アプローチ—

Practical Introduction to Machine Learning in Web Browsers:
A Step-by-Step Learning Approach Using ml5.js

河合 勝彦

名古屋市立大学大学院経済学研究科

kkawai@econ.nagoya-cu.ac.jp

2026 年 1 月 11 日

概要

本稿は、Web ブラウザ上で動作する機械学習ライブラリ ml5.js を用いた実践的な学習ガイドである。ml5.js は TensorFlow.js を基盤とし、画像分類、姿勢推定、音声認識などの機械学習機能を JavaScript の簡潔な API で提供する。本ガイドでは、30 個の段階的なサンプルプログラムを通じて、初学者から上級者までが機械学習の概念と Web 開発技術を同時に習得できるカリキュラムを構成した。

各サンプルは単一の HTML ファイルで完結し、外部サーバーやビルドツールを必要としない設計となっている。これにより、学習者は環境構築の障壁なく、即座に機械学習アプリケーションの開発を体験できる。初級編では各モデルの基本的な使用法を、中級編ではゲームやニューラルネットワーク訓練などの応用的な実装を、上級編ではカスタム分類器やマルチモーダルシステムの構築を扱う。

さらに、大規模言語モデル（LLM）を活用した学習支援のためのプロンプト集を付録として収録し、現代的な AI 支援学習のアプローチも提示する。本ガイドは、機械学習の民主化とクライアントサイド AI の可能性を探求する教育・研究資料として位置づけられる。

キーワード

和文: 機械学習、深層学習、Web プログラミング、JavaScript、画像認識、姿勢推定、音声認識、ニューラルネットワーク、クライアントサイド AI、プログラミング教育

英文: Machine Learning, Deep Learning, Web Programming, JavaScript, Image Classification, Pose Estimation, Speech Recognition, Neural Network, Client-side AI, Programming Education

目次

はじめに

ml5.js とは

ml5.js は、ブラウザ上で機械学習を簡単に利用できる JavaScript ライブラリです。TensorFlow.js をベースに構築されており、複雑な機械学習の知識がなくても、画像認識、ポーズ検出、音声分類などの高度な機能を数行のコードで実装できます。

ml5.js の特徴は、その「親しみやすさ」にあります。従来、機械学習を Web アプリケーションに組み込むには、モデルのアーキテクチャ設計、訓練データの準備、パラメータのチューニングなど、専門的な知識が必要でした。ml5.js は、これらの複雑さを抽象化し、事前学習済みモデルを簡単に呼び出せる API を提供しています。

また、ml5.js はクライアントサイドで完結するため、サーバーにデータを送信する必要がありません。これはプライバシーの観点からも重要であり、カメラやマイクの入力をローカルで処理できます。

本ガイドの目的

本ガイドは、30 個の実践的なサンプルを通じて、ml5.js の使い方を段階的に習得することを目的としています。各サンプルは単独で動作する完結した HTML ファイルであり、ソースコードを読み、実行し、改変することで、機械学習の概念と Web 開発のテクニックを同時に学ぶことができます。

単にコードをコピーするのではなく、「なぜこのように書くのか」「どのような仕組みで動いているのか」を理解することで、オリジナルのアプリケーションを開発できる力を身につけましょう。

本ガイドの構成

- **第 1 章：初級編（サンプル 01〜10）** – 各モデルの基本的な使い方を学びます。画像分類、ポーズ検出、顔検出、音声分類など、ml5.js が提供する主要な機能を一通り体験します。
- **第 2 章：中級編（サンプル 11〜20）** – 複数の機能を組み合わせたインタラクティブなアプリケーションを作成します。ゲーム、お絵かき、ニューラルネットワークの訓練など、より実践的な内容に取り組みます。
- **第 3 章：上級編（サンプル 21〜30）** – カスタムモデルの訓練、複数人の追跡、複合的なシステムなど、高度なトピックを扱います。実際のアプリケーション開発に直結するスキルを習得します。
- **付録 A** – ml5.js 関数・メソッド一覧

環境構築

必要なもの

- **モダンな Web ブラウザ**（Chrome、Firefox、Edge 等）– 最新版を推奨します。特に Chrome は Web カメラやマイクのアクセスが安定しています。
- **ウェブカメラ**（ポーズ検出系サンプル用）– 内蔵カメラでも外付けカメラでも構いません。
- **マイク**（音声分類サンプル用）– 内蔵マイクで十分です。
- **ローカルサーバー**（推奨）– セキュリティ上の理由から、カメラやマイクを使用するサンプルはローカルサーバー経由でアクセスする必要があります。

サンプルの実行方法

```
# リポジトリをクローン
git clone https://github.com/kkawailab/kklab-ml5js-samples.git

# ローカルサーバーを起動
cd kklab-ml5js-samples
npx serve .
# または
python -m http.server 8000
```

ブラウザで `http://localhost:3000` または `http://localhost:8000` にアクセスしてサンプルを実行します。

注意

ファイルを直接ブラウザで開く（`file://`プロトコル）と、セキュリティ制限によりカメラやマイクにアクセスできない場合があります。必ずローカルサーバーを経由してください。

共通パターン

すべてのサンプルは以下の構造で統一されています。この構造を理解しておくことで、コードの読解がスムーズになります。

```
<!-- 1. ml5.jsをCDNから読み込み -->
<script src="https://unpkg.com/ml5@1/dist/ml5.min.js"></script>

<script>
// 2. グローバル変数の宣言
let video, canvas, ctx;
let model;
```

```
8
9  // 3. 初期化関数
10 async function setup() {
11     // カメラの取得
12     video = await getVideo();
13     // ml5.jsモデルの読み込み
14     model = await ml5.bodyPose('MoveNet');
15     // 検出ループの開始
16     detect();
17 }
18
19 // 4. 検出ループ
20 async function detect() {
21     const results = await model.detect(video);
22     draw(results);
23     requestAnimationFrame(detect);
24 }
25
26 // 5. ページ読み込み時に初期化
27 window.addEventListener('load', setup);
28 </script>
```

実装のポイント

左右反転について

ウェブカメラの映像は通常「鏡像」として表示します。これにより、ユーザーが右手を上げると画面上でも右側の手が上がるため、直感的に操作できます。`ctx.scale(-1, 1)` を使用して X 軸を反転させています。

倫理・プライバシーへの配慮

本ガイドのサンプルは Web カメラやマイクを使用します。教育現場で使用する際は、以下の点に配慮してください。

注意

授業での注意事項

- **事前の同意**: カメラやマイクを使用する前に、受講者に目的を説明し、同意を得てください。「このサンプルではカメラ映像を使いますが、録画・保存はしません」など、一言添えると安心です。
- **録画・録音の禁止**: 本サンプルは学習目的であり、映像や音声の保存機能は含まれていません。録画・録音が必要な場合は、別途明確な同意を取得してください。

- **参加の任意性**: カメラに映ることを望まない受講者には、他の受講者のデモを見学する、または静止画像を使うサンプル（01, 09 等）で学ぶ選択肢を提供してください。

ヒント

誤認識への対応

機械学習モデルは完璧ではなく、誤認識が起こることがあります。特に以下の点を受講者に伝えておくと良いでしょう。

- 照明条件や背景によって精度が変わることがある
- 特定の肌色や体型で精度が低下する可能性がある（訓練データの偏り）
- デモが失敗しても問題ない—それも学びの一部である

第 1 章

初級編：基本を学ぶ

この章の到達目標と所要時間

到達目標

- ml5.js の基本的な API パターン（初期化→検出→描画）を理解できる
- 画像分類・ポーズ検出・顔検出・音声分類の各モデルを使用できる
- async/await を使った非同期処理の基本を理解できる

推奨所要時間

- 全サンプル通し：90 分 ×3〜4 回
- 重点サンプル（01, 03, 06, 07）のみ：90 分 ×2 回

この章では、ml5.js が提供する主要なモデルの基本的な使い方を学びます。各サンプルは独立しており、興味のあるものから始めることができます。

1.1 サンプル 01：画像分類 - 基本

1.1.1 概要

このサンプルは、ml5.js を使った機械学習の最初の一歩として最適な入門例です。事前学習済みの MobileNet モデルを使用して、静止画像に何が写っているかを分類（識別）します。

MobileNet は、Google が開発した軽量な画像分類モデルです。ImageNet の 1000 クラス分類タスク（ILSVRC）で訓練されており、日常的な物体（動物、乗り物、食べ物など）を高い精度で認識できます。「軽量」というのは、モデルのサイズが小さく、スマートフォンやブラウザでも高速に動作することを意味します。

このサンプルを通じて、機械学習における「推論（inference）」の基本的な流れを理解できます。推論とは、訓練済みのモデルに新しいデータを入力し、予測結果を得るプロセスです。

1.1.2 学べること

- **ml5.imageClassifier() の基本的な使い方** – モデルの初期化から推論までの一連の流れを学びます。async/await を使った非同期処理の書き方も重要なポイントです。
- **事前学習モデルの概念** – 自分でモデルを訓練しなくても、既に訓練されたモデルを利用できることを理解します。これにより、少ないコードで高度な機能を実現できます。
- **分類結果の構造** – 結果は「ラベル（何であるか）」と「確信度（どのくらい確かか）」のペアで返されます。確信度は 0 から 1 の値で、1 に近いほど確信が高いことを示します。

1.1.3 動作の流れ

1. ページが読み込まれると、MobileNet モデルをダウンロードして初期化します（数秒かかります）
2. ユーザーが画像を選択するか、デフォルトの画像が表示されます
3.「分類」ボタンをクリックすると、画像がモデルに渡されます
4. モデルは画像を分析し、最も可能性の高いカテゴリを上位 5 件返します
5. 結果がラベルと確信度のリストとして画面に表示されます

1.1.4 主要なコード

```
// モデルの初期化
// 'MobileNet'を指定することで、事前学習済みモデルが
// 自動的にダウンロードされます
// awaitを使用しているため、モデルの準備が完了するまで待機します
const classifier = await ml5.imageClassifier('MobileNet');

// 画像を分類
// imageElementはHTML上の<img>要素を指します
// classify()メソッドは非同期で実行され、結果をPromiseで返します
const results = await classifier.classify(imageElement);

// 結果の表示（上位5件）
// resultsは配列で、各要素はlabelとconfidenceを持ちます
results.forEach(result => {
    // labelは英語のカテゴリ名（例: "golden retriever"）
    // confidenceは0〜1の数値（例: 0.85は85%の確信度）
    console.log(`${result.label}: ${(result.confidence * 100).toFixed
        (1)}%`);
});
```

ヒント

MobileNet は英語のラベルを返します。日本語で表示したい場合は、翻訳用のマッピングオブジェクトを用意するか、翻訳 API を利用することを検討してください。

実装のポイント

確信度の解釈

確信度が高いからといって、必ずしも正しいとは限りません。モデルが見たことのないタイプの画像では、高い確信度で誤った結果を返すことがあります。これは機械学習の重要な特性であり、結果を鵜呑みにせず、常に批判的に評価することが大切です。

拡張のヒント

拡張のヒント：複数画像の一括分類

複数の画像を順番に分類し、結果を比較表示する機能を追加できます。

```
const images = document.querySelectorAll('.target-image');
const allResults = [];
for (const img of images) {
    const results = await classifier.classify(img);
    allResults.push({
        image: img.src,
        topLabel: results[0].label,
        confidence: results[0].confidence
    });
}
```

1.2 サンプル 02：リアルタイム画像分類

1.2.1 概要

サンプル 01 では静止画像を分類しましたが、このサンプルではウェブカメラの映像をリアルタイムで分類し続けます。カメラに映るものが変わると、即座に分類結果が更新されます。

リアルタイム処理は、機械学習アプリケーションの魅力的な機能の一つです。ユーザーがカメラの前で物を動かすと、その物体が何であるかが瞬時に表示されます。これは、物体認識アプリ、アクセシビリティツール、教育用アプリケーションなど、様々な用途に応用できます。

このサンプルでは、JavaScript の重要な概念である「アニメーションループ」も学びます。`requestAnimationFrame` を使用して、ブラウザのリフレッシュレートに同期した滑らかな更新を実現しています。

1.2.2 学べること

- **ウェブカメラ映像の取得** – `navigator.mediaDevices.getUserMedia()` を使用して、ユーザーのカメラにアクセスします。ブラウザはユーザーに許可を求め、許可された場合のみ映像を取得できます。
- **継続的な推論ループの構築** – 一度だけでなく、継続的に推論を実行するためのループ処理を実装します。
- **パフォーマンスの考慮** – リアルタイム処理では、推論速度と UI 更新のバランスが重要です。

1.2.3 動作の流れ

1. ページ読み込み時に、カメラへのアクセス許可をリクエスト
2. ユーザーが許可すると、カメラ映像がビデオ要素に表示される
3. MobileNet モデルを初期化
4. `requestAnimationFrame` を使って、毎フレーム分類を実行

1.2.4 主要なコード

以下のコードは、ウェブカメラの映像を継続的に分類するための 2 つの主要な関数で構成されています。

`setupCamera()` 関数は、ブラウザの `getUserMedia` API を使用してカメラにアクセスします。`video` オプションで解像度と向き（`facingMode: 'user'` で前面カメラ）を指定し、取得したストリームを `video` 要素に設定します。Promise を返すことで、映像の準備が完了するまで待機できるようにしています。

`classifyFrame()` 関数は、分類と表示を繰り返すループを構成します。`classifier.classify(video)` でビデオフレームを分類し、結果を `displayResult()` で画面に表示した後、`requestAnimationFrame` で自分自身を呼び出すことで、ブラウザのリフレッシュレートに同期した滑らかな更新を実現しています。

```
// ウェブカメラの取得
async function setupCamera() {
    const stream = await navigator.mediaDevices.getUserMedia({
        video: { width: 640, height: 480, facingMode: 'user' }
    });
    video.srcObject = stream;
    return new Promise(resolve => {
        video.onloadedmetadata = () => {
            video.play();
            resolve(video);
        };
    });
```

```
}

// 連続分類ループ
async function classifyFrame() {
    const results = await classifier.classify(video);
    displayResult(results[0]);
    requestAnimationFrame(classifyFrame);
}

function displayResult(result) {
    labelElement.textContent = result.label;
    confidenceElement.textContent =
        `${(result.confidence * 100).toFixed(1)}%`;
}
```

注意

無限ループに注意

`requestAnimationFrame` を使ったループは、明示的に停止しない限り永続的に実行されます。ページを離れる際やカメラを停止する際は、適切にループを終了させる処理を追加してください。

拡張のヒント

拡張のヒント：分類履歴の記録

過去の分類結果を配列に保存し、時系列で表示できます。

```
const history = [];
function recordResult(result) {
    history.push({
        label: result.label,
        confidence: result.confidence,
        timestamp: Date.now()
    });
    if (history.length > 100) history.shift();
}
```

1.3 サンプル 03：BodyPose - 基本

1.3.1 概要

このサンプルでは、MoveNet モデルを使用して、人体の姿勢（ポーズ）を検出します。カメラに映る人物の 17 個の関節位置（キーポイント）をリアルタイムで追跡し、画面上に点として表示します。

ポーズ検出は、フィットネスアプリ、ゲーム、ジェスチャー認識、ダンス練習など、幅広い応用が

可能な技術です。従来、このような機能を実装するには専用のハードウェア（モーションキャプチャ機器など）が必要でしたが、ml5.js を使えば一般的なウェブカメラだけで実現できます。

MoveNet は、Google が開発した高速・高精度なポーズ検出モデルです。リアルタイム処理に最適化されており、一般的な PC やスマートフォンでも滑らかに動作します。

1.3.2 学べること

- **ml5.bodyPose() の使い方** – ポーズ検出モデルの初期化と、detect() メソッドによる検出を学びます。
- **17 個のキーポイントの意味と位置** – 鼻、目、耳、肩、肘、手首、腰、膝、足首など、各キーポイントが体のどの部位に対応するかを理解します。
- **信頼度（confidence）による検出品質の判定** – 各キーポイントには信頼度が付与されています。遮蔽された部位や画面外の部位は信頼度が低くなります。

1.3.3 17 個のキーポイント

BodyPose（MoveNet）は、以下の 17 個のキーポイントを検出します。

番号	部位	番号	部位
0	鼻	9	左手首
1	左目	10	右手首
2	右目	11	左腰（股関節）
3	左耳	12	右腰（股関節）
4	右耳	13	左膝
5	左肩	14	右膝
6	右肩	15	左足首
7	左肘	16	右足首
8	右肘		

表 1.1 BodyPose のキーポイント一覧

ヒント

「左」「右」は画像の視点ではなく、その人物から見た左右を指します。鏡像表示を行っている場合、画面上では逆に見えます。

1.3.4 主要なコード

以下のコードは、BodyPose モデルの初期化と検出ループの実装を示しています。

まず `ml5.bodyPose('MoveNet')` で MoveNet モデルを非同期で読み込みます。モデルの読み込みには数秒かかる場合があるため、`await` で完了を待ちます。

`detect()` 関数では、`bodyPose.detect(video)` でビデオフレームからポーズを検出します。結果は配列で返され、各要素が検出された人物に対応します。`results[0].keypoints` で最初の人物の 17 個のキーポイントにアクセスできます。

各キーポイントには x、y 座標と `confidence`（信頼度）が含まれます。信頼度が閾値（この例では 0.3）を超えるキーポイントのみを円として描画することで、遮蔽された部位や画面外の部位を除外しています。`getColorForKeypoint()` 関数では、キーポイントの番号に応じて色を変えることで、顔・上半身・下半身を視覚的に区別しています。

```
// BodyPoseモデルの初期化
const bodyPose = await ml5.bodyPose('MoveNet');

// 検出ループ
async function detect() {
    const results = await bodyPose.detect(video);
    if (results.length > 0) {
        const keypoints = results[0].keypoints;
        keypoints.forEach((kp, index) => {
            if (kp.confidence > 0.3) {
                ctx.beginPath();
                ctx.arc(kp.x, kp.y, 8, 0, Math.PI * 2);
                ctx.fillStyle = getColorForKeypoint(index);
                ctx.fill();
            }
        });
    }
    requestAnimationFrame(detect);
}

function getColorForKeypoint(index) {
    if (index <= 4) return '#FF6B6B';       // 顔: 赤
    if (index <= 10) return '#4ECDC4';      // 上半身: シアン
    return '#45B7D1';                        // 下半身: 青
}
```

実装のポイント

信頼度の閾値について

信頼度の閾値（この例では 0.3）は、アプリケーションの目的に応じて調整してください。低い閾値ではより多くのキーポイントが表示されますが誤検出も増えます。高い閾値では確実なキーポイントのみが表示されます。

拡張のヒント

拡張のヒント：キーポイントの詳細表示

各キーポイントの座標と信頼度をテーブル表示できます。

```
function displayKeypointInfo(keypoints) {
    keypoints.forEach(kp => {
        console.log(`${kp.name}: (${kp.x.toFixed(0)}, ${kp.y.toFixed
            (0)}) - ${(kp.confidence * 100).toFixed(1)}%`);
    });
}
```

1.4 サンプル 04：BodyPose - スケルトン描画

1.4.1 概要

サンプル 03 ではキーポイントを個別の点として表示しましたが、このサンプルでは点と点を線で結んで「骨格（スケルトン）」を描画します。これにより、人体の構造がより直感的に可視化されます。

スケルトン描画は、モーションキャプチャ、ダンス練習アプリ、フィットネストラッカーなど、多くのアプリケーションで使用される基本的な視覚化手法です。

1.4.2 学べること

- **キーポイント間の接続定義** – どのキーポイント同士を線で結ぶかを定義します。
- **Canvas API を使った線描画** – `beginPath()`, `moveTo()`, `lineTo()`, `stroke()` を使った線の描画方法を学びます。
- **条件付き描画** – 両端のキーポイントの信頼度が十分な場合のみ線を描画します。

1.4.3 骨格の接続構造

- **顔**: 鼻 - 左目、鼻 - 右目、左目 - 左耳、右目 - 右耳
- **胴体**: 左肩 - 右肩、左肩 - 左腰、右肩 - 右腰、左腰 - 右腰
- **腕**: 肩 - 肘 - 手首
- **脚**: 腰 - 膝 - 足首

1.4.4 主要なコード

以下のコードは、キーポイント間を線で結んでスケルトン（骨格）を描画する仕組みを示しています。

`connections` 配列は、どのキーポイント同士を線で結ぶかを定義しています。各要素は **[開始点，**

終了点] の形式で、キーポイントの番号を指定します。例えば [5, 7] は「左肩（5 番）から左肘（7 番）へ線を引く」ことを意味します。

drawSkeleton() 関数では、Canvas API の線描画機能を使用します。strokeStyle で線の色、lineWidth で太さ、lineCap で線端の形状を設定します。各接続について、両端のキーポイントの信頼度を確認し、両方が閾値を超える場合のみ moveTo() で始点に移動し、lineTo() で終点まで線を引き、stroke() で描画を確定します。

```
// 骨格の接続定義
const connections = [
    [0, 1], [0, 2], [1, 3], [2, 4],  // 顔
    [5, 6], [5, 11], [6, 12], [11, 12],  // 胴体
    [5, 7], [7, 9],   // 左腕
    [6, 8], [8, 10],  // 右腕
    [11, 13], [13, 15],  // 左脚
    [12, 14], [14, 16]   // 右脚
];

function drawSkeleton(keypoints) {
    ctx.strokeStyle = '#00ff00';
    ctx.lineWidth = 3;
    ctx.lineCap = 'round';

    connections.forEach(([i, j]) => {
        const a = keypoints[i];
        const b = keypoints[j];
        if (a.confidence > 0.3 && b.confidence > 0.3) {
            ctx.beginPath();
            ctx.moveTo(a.x, a.y);
            ctx.lineTo(b.x, b.y);
            ctx.stroke();
        }
    });
}
```

拡張のヒント

拡張のヒント：部位ごとの色分け

体の部位ごとに異なる色で描画すると視覚的にわかりやすくなります。

```
const bodyParts = {
    face: { connections: [[0,1], [0,2]], color: '#FF6B6B' },
    torso: { connections: [[5,6], [5,11], [6,12]], color: '#4ECDC4'
        },
    arms: { connections: [[5,7], [7,9], [6,8], [8,10]], color: '#45
        B7D1' },
    legs: { connections: [[11,13], [13,15], [12,14], [14,16]], color
```

```
            : '#96CEB4' }
};
```

1.5 サンプル 05：HandPose - 手の検出

1.5.1 概要

このサンプルでは、HandPose モデルを使用して手の 21 個のキーポイントを検出します。手首と各指の 4 つの関節（計 5 本 ×4 ＋手首 1 ＝ 21 個）をリアルタイムで追跡できます。

手の検出は、ジェスチャー認識、手話翻訳、バーチャルキーボード、お絵かきアプリなど、直感的なインタラクションを実現するための基盤技術です。

1.5.2 学べること

- **ml5.handPose() の使い方** – 手検出モデルの初期化と使用方法を学びます。
- **21 個のキーポイントの構造** – 手首（1 個）＋ 5 本の指 ×4 関節（20 個）の配置を理解します。
- **指先の識別方法** – キーポイントの番号から、どの指のどの関節かを特定する方法を学びます。

1.5.3 21 個のキーポイントの構造

- **0**: 手首
- **1-4**: 親指（付け根から指先へ）
- **5-8**: 人差し指
- **9-12**: 中指
- **13-16**: 薬指
- **17-20**: 小指

各指の最後のキーポイント（4, 8, 12, 16, 20）が指先に対応します。

1.5.4 主要なコード

以下のコードは、HandPose モデルによる手の検出と、指の骨格構造の定義を示しています。

`ml5.handPose()` でモデルを初期化します。引数なしで呼び出すとデフォルト設定が使用されます。

`detect()` 関数では、検出結果を `forEach` で処理します。複数の手が検出される可能性があるため、各 `hand` オブジェクトに対して骨格と点を描画します。

`fingerConnections` 配列は、手首から各指先までの接続を定義しています。各配列が 1 本の指に対応し、手首（0 番）から指先まで順番にキーポイント番号を並べています。例えば人差し指は `[0,`

5, 6, 7, 8] で、手首→付け根→第 2 関節→第 1 関節→指先の順に接続します。指先は各配列の最後の番号（4, 8, 12, 16, 20）に対応します。

```
const handPose = await ml5.handPose();

async function detect() {
    const results = await handPose.detect(video);
    results.forEach(hand => {
        drawHandSkeleton(hand.keypoints);
        drawHandKeypoints(hand.keypoints);
    });
    requestAnimationFrame(detect);
}

const fingerConnections = [
    [0, 1, 2, 3, 4],      // 親指
    [0, 5, 6, 7, 8],      // 人差し指
    [0, 9, 10, 11, 12],   // 中指
    [0, 13, 14, 15, 16],  // 薬指
    [0, 17, 18, 19, 20]   // 小指
];
```

拡張のヒント

拡張のヒント：指先追跡エフェクト

人差し指の軌跡を残すエフェクトを追加できます。

```
const trail = [];
function updateTrail(hand) {
    const fingerTip = hand.keypoints[8];
    trail.push({x: fingerTip.x, y: fingerTip.y});
    if (trail.length > 50) trail.shift();
}
```

1.6 サンプル 06：FaceMesh - 顔のランドマーク

1.6.1 概要

FaceMesh は、顔の 468〜478 個のランドマーク（特徴点）を検出するモデルです。目、眉、鼻、口、顔の輪郭など、顔のあらゆる部分の詳細な位置情報を取得できます。基本で 468 点、refineLandmarks オプションを有効にすると虹彩の 10 点が追加され 478 点になります。

この技術は、顔フィルター、表情分析、視線追跡、バーチャルメイクアップなど、多くのアプリケーションで使用されています。

1.6.2 学べること

- **ml5.faceMesh() の使い方** – 顔検出モデルの初期化と、大量のキーポイントの処理方法を学びます。
- **主要な顔のポイントの特定** – 目的に応じて重要なポイントを選択して使用します。

1.6.3 主要な顔のポイント

- **1**: 鼻先
- **33**: 左目の外側、**263**: 右目の外側
- **61, 291**: 口の両端
- **10**: 額の中心、**152**: あご先

1.6.4 主要なコード

以下のコードは、FaceMesh モデルによる顔のランドマーク検出と描画の基本パターンを示しています。

`ml5.faceMesh()` でモデルを初期化します。FaceMesh は数百個という大量のキーポイントを検出するため、モデルサイズはやや大きめです。

`detect()` 関数では、検出結果の有無を確認してから処理を行います。FaceMesh は顔が検出されない場合に空の配列を返すため、`results.length > 0` でチェックしています。検出された顔の `keypoints` 配列には 468〜478 個のポイントが含まれ、それぞれに `x`、`y` 座標があります。

ここでは全ポイントを半径 1 ピクセルの小さな円として描画しています。全ポイントを描画すると顔の輪郭や特徴が浮かび上がります。実際のアプリケーションでは、目的に応じて特定のポイント（目、鼻、口など）だけを使用するのが一般的です。

```
const faceMesh = await ml5.faceMesh();

async function detect() {
    const results = await faceMesh.detect(video);
    if (results.length > 0) {
        const keypoints = results[0].keypoints;
        // 全ポイントを小さな点で描画
        keypoints.forEach(kp => {
            ctx.beginPath();
            ctx.arc(kp.x, kp.y, 1, 0, Math.PI * 2);
            ctx.fill();
        });
    }
    requestAnimationFrame(detect);
}
```

拡張のヒント

拡張のヒント：顔の輪郭描画

特定のポイントを結んで顔の輪郭を描画できます。

```
const faceContour = [10, 338, 297, 332, 284, 251, 389, 356, ...];
function drawContour(keypoints, indices) {
    ctx.beginPath();
    indices.forEach((idx, i) => {
        const kp = keypoints[idx];
        i === 0 ? ctx.moveTo(kp.x, kp.y) : ctx.lineTo(kp.x, kp.y);
    });
    ctx.closePath();
    ctx.stroke();
}
```

1.7 サンプル 07：音声分類 - 基本

1.7.1 概要

このサンプルでは、マイクからの音声入力をリアルタイムで分類します。SpeechCommands18w モデルを使用し、18 種類の英語の音声コマンドを認識できます。

1.7.2 学べること

- **ml5.soundClassifier() の使い方** – 音声分類モデルの初期化と使用方法を学びます。
- **マイク入力の取得** – ブラウザでマイクにアクセスする方法を理解します。

1.7.3 認識可能な 18 コマンド

zero, one, two, three, four, five, six, seven, eight, nine, yes, no, up, down, left, right, go, stop

1.7.4 主要なコード

以下のコードは、音声分類の初期化とコールバックパターンを示しています。

`ml5.soundClassifier('SpeechCommands18w')` でモデルを初期化します。'SpeechCommands18w' は 18 種類の英語音声コマンドを認識できる事前学習済みモデルです。

音声分類は他のモデルと異なり、コールバック関数を使った連続分類パターンを採用しています。`classifier.classify(gotResult)` を一度呼び出すと、内部でマイク入力を監視し続け、分類結果が得られるたびに `gotResult` 関数が自動的に呼び出されます。

`gotResult()` 関数では、結果の存在確認を行ってから `results[0]` で最も確信度の高い結果を取

得します。label に認識されたコマンド名、confidence に確信度が含まれます。

```
const classifier = await ml5.soundClassifier('SpeechCommands18w');

function startClassification() {
    classifier.classify(gotResult);
}

function gotResult(results) {
    if (results && results.length > 0) {
        const topResult = results[0];
        displayResult(topResult.label, topResult.confidence);
    }
}
```

拡張のヒント

拡張のヒント：音声コマンドで UI 操作

```
function handleCommand(label, confidence) {
    if (confidence < 0.8) return;
    switch (label) {
        case 'up': window.scrollBy(0, -100); break;
        case 'down': window.scrollBy(0, 100); break;
    }
}
```

1.8 サンプル 08：スクワットカウンター

1.8.1 概要

このサンプルは、BodyPose を使った実用的なアプリケーションの例です。膝の屈伸を検出してスクワットの回数を自動的にカウントします。

1.8.2 学べること

- **3 点間の角度計算** – Math.atan2 を使って関節の角度を計算します。
- **状態遷移による動作判定** –「立っている」「しゃがんでいる」の 2 状態を定義し、遷移をカウントします。

1.8.3 主要なコード

以下のコードは、関節角度の計算と状態遷移によるスクワット検出の仕組みを示しています。

`calculateAngle()` 関数は、3 点（a、b、c）から中間点 b での角度を計算します。`Math.atan2()` は 2 点間のベクトルの角度をラジアンで返す関数で、c-b 方向と a-b 方向の角度差を計算することで、b 点での曲がり具合を求めています。結果をラジアンから度に変換し、180 度を超える場合は 360 度から引くことで、常に 0〜180 度の範囲に収めています。

`updateSquatCount()` 関数では、状態遷移を使ってスクワット回数をカウントします。`state` 変数は「立っている（up）」か「しゃがんでいる（down）」かを保持します。膝の角度（腰-膝-足首で計算）が 90 度未満になったら「down」に、160 度以上に戻ったら「up」に遷移し、「down」から「up」への遷移時にカウントを増やします。この状態遷移パターンにより、中途半端な動きや揺れによる誤カウントを防いでいます。

```
function calculateAngle(a, b, c) {
    const radians = Math.atan2(c.y - b.y, c.x - b.x)
                  - Math.atan2(a.y - b.y, a.x - b.x);
    let angle = Math.abs(radians * 180 / Math.PI);
    if (angle > 180) angle = 360 - angle;
    return angle;
}

let state = 'up';
let count = 0;

function updateSquatCount(keypoints) {
    const hip = keypoints[12];
    const knee = keypoints[14];
    const ankle = keypoints[16];

    const kneeAngle = calculateAngle(hip, knee, ankle);

    if (kneeAngle < 90 && state === 'up') {
        state = 'down';
    } else if (kneeAngle > 160 && state === 'down') {
        state = 'up';
        count++;
    }
}
```

拡張のヒント

拡張のヒント：複数エクササイズ対応

```
const exercises = {
    squat: { joints: [12, 14, 16], threshold: 90 },
    pushup: { joints: [6, 8, 10], threshold: 90 }
};
```

1.9 サンプル 09：Image Classifier - 画像アップロード

1.9.1 概要

ドラッグ&ドロップまたはファイル選択で画像をアップロードし、分類します。ファイル入力処理と UI の連携を学びます。

1.9.2 学べること

- **FileReader API による画像読み込み**
- **ドラッグ＆ドロップのイベント処理**
- **分類結果の視覚的表示**

1.9.3 主要なコード

以下のコードは、ドラッグ&ドロップによるファイル受け取りと、FileReader API を使った画像の読み込み・分類処理を示しています。

`'drop'` イベントリスナーでは、まず `e.preventDefault()` でブラウザのデフォルト動作（ファイルを新しいタブで開くなど）を防止します。`e.dataTransfer.files[0]` でドロップされた最初のファイルを取得し、`file.type.startsWith('image/')` で画像ファイルかどうかを確認してから処理を行います。

`processImage()` 関数では、`FileReader` を使ってファイルを Data URL 形式で読み込みます。`readAsDataURL()` は非同期処理のため、`onload` コールバックで読み込み完了を待ちます。読み込んだ Data URL を新しい `Image` オブジェクトの `src` に設定し、画像の読み込みが完了したら `classifier.classify()` で分類を実行します。

```
dropArea.addEventListener('drop', async (e) => {
    e.preventDefault();
    const file = e.dataTransfer.files[0];
    if (file && file.type.startsWith('image/')) {
        await processImage(file);
    }
});

async function processImage(file) {
    const reader = new FileReader();
    reader.onload = async (e) => {
        const img = new Image();
        img.src = e.target.result;
        img.onload = async () => {
            const results = await classifier.classify(img);
            displayResults(results);
```

```
            };
        };
        reader.readAsDataURL(file);
    }
```

1.10 サンプル 10：顔に絵文字オーバーレイ

1.10.1 概要

FaceMesh で検出した顔の位置に絵文字を重ねて表示します。SNS のフィルター機能の基本的な仕組みを学べます。

1.10.2 学べること

- **特定のキーポイントの活用** – 目や鼻の位置を使ってオーバーレイの配置位置を決定します。
- **動的スケーリング** – 顔のサイズに応じて絵文字のサイズを調整します。

1.10.3 主要なコード

以下のコードは、顔の位置とサイズに合わせて絵文字を動的に配置する処理を示しています。

`drawEmojiOverlay()` 関数では、まず左目（33 番）と右目（263 番）のキーポイントを取得します。この 2 点間の距離（ユークリッド距離）を計算することで、顔の大きさの指標を得ています。カメラからの距離に応じて目の間隔は変化するため、これを基準にすることで顔のサイズに合った絵文字サイズを決定できます。

絵文字のサイズは目の距離の 2.5 倍に設定しています。この倍率は絵文字の種類や見た目のバランスに応じて調整します。配置位置は鼻先（1 番）を基準にし、`ctx.textAlign = 'center'` で水平方向の中央揃えを設定しています。`ctx.font` でサイズを指定し、`ctx.fillText()` で絵文字を描画します。

```
async function drawEmojiOverlay(keypoints) {
    const leftEye = keypoints[33];
    const rightEye = keypoints[263];
    const eyeDistance = Math.sqrt(
        Math.pow(rightEye.x - leftEye.x, 2) +
        Math.pow(rightEye.y - leftEye.y, 2)
    );
    const emojiSize = eyeDistance * 2.5;
    const nose = keypoints[1];

    ctx.font = `${emojiSize}px serif`;
    ctx.textAlign = 'center';
    ctx.fillText(currentEmoji, nose.x, nose.y);
}
```

第 2 章

中級編：機能を組み合わせる

この章の到達目標と所要時間

到達目標

- 複数の ml5.js モデルを組み合わせたアプリケーションを設計できる
- ゲームループ・衝突検出・状態管理の基本パターンを実装できる
- ml5.neuralNetwork() を使ってカスタムモデルを訓練できる

推奨所要時間

- 全サンプル通し：90 分 ×4〜5 回
- 重点サンプル（11, 14, 17, 18）のみ：90 分 ×2〜3 回

この章では、初級編で学んだ基本を組み合わせて、より実践的なアプリケーションを作成します。ゲーム開発、インタラクティブアート、ニューラルネットワークの訓練など、創造的なプロジェクトに取り組みます。

2.1 サンプル 11：フルーツキャッチゲーム

2.1.1 概要

このサンプルでは、BodyPose を使った体験型ゲームを作成します。画面上部から落ちてくるフルーツを、両手を動かしてキャッチするゲームです。マウスやキーボードを使わず、体の動きだけでゲームを操作する新しい体験を作り出せます。

ゲーム開発の基本要素（ゲームループ、衝突検出、スコア管理、難易度調整）と、ポーズ検出を組み合わせる方法を学びます。

2.1.2 学べること

- **ゲームループの構築** – 状態更新、描画、入力処理を毎フレーム実行
- **衝突検出** – 手の位置とフルーツの距離を計算して判定

- **スコアとライフの管理** – ゲームの進行状況を追跡
- **難易度の動的調整** – スコアに応じて落下速度を変更

2.1.3 主要なコード

以下のコードは、ゲームオブジェクトの管理と衝突検出の基本的な実装パターンを示しています。

Fruit クラスは、画面上部から落下するフルーツを表現します。コンストラクタで横位置をランダムに設定し、縦位置は画面上部の外側（-30）から開始します。落下速度は基本速度 2 に難易度に応じた加算を行い、絵文字配列からランダムに選んだフルーツを設定します。update() メソッドで毎フレーム位置を更新します。

checkCollision() 関数は、2 点間のユークリッド距離を計算し、閾値（50 ピクセル）以内であれば衝突と判定します。この「円同士の当たり判定」は最もシンプルな衝突検出アルゴリズムです。

gameLoop() 関数では、BodyPose から取得した左手首（9 番）と右手首（10 番）の位置を使って、全フルーツとの衝突判定を行います。どちらかの手がフルーツに触れたらスコアを加算し、フルーツを削除します。

```
class Fruit {
    constructor() {
        this.x = Math.random() * canvas.width;
        this.y = -30;
        this.speed = 2 + difficulty * 0.5;
        this.emoji = [' ', ' ', ' ', ' '][Math.floor(Math.random() * 4)];
    }
    update() { this.y += this.speed; }
}

function checkCollision(fruit, hand) {
    const distance = Math.sqrt(
        Math.pow(fruit.x - hand.x, 2) + Math.pow(fruit.y - hand.y, 2)
    );
    return distance < 50; // 当たり判定の半径
}

function gameLoop(keypoints) {
    const leftHand = keypoints[9];   // 左手首
    const rightHand = keypoints[10]; // 右手首

    fruits.forEach(fruit => {
        if (checkCollision(fruit, leftHand) || checkCollision(fruit,
            rightHand)) {
            score += 10;
            // フルーツを削除
        }
    });
```

```
28 }
```

拡張のヒント

拡張のヒント：パワーアップアイテム
スローモーション、ダブルポイント、ライフ回復などの特殊アイテムを追加できます。

2.2 サンプル 12：手で描くお絵かき

2.2.1 概要

人差し指で空中に絵を描くアプリケーションです。指の伸び状態を判定し、人差し指だけが伸びているときに描画モードになります。これにより、描画の ON/OFF を直感的に切り替えられます。

2.2.2 学べること

- **指の伸び/曲がり判定** – 指先と関節の位置関係から状態を判定
- **2 層 Canvas の使用** – 映像用と描画用を分離して重ね合わせ
- **スムーズな線描画** – quadraticCurveTo で滑らかな曲線を実現

2.2.3 主要なコード

以下のコードは、指の伸び状態の判定と、それを使った描画モードの切り替えを示しています。

`isFingerExtended()` 関数は、指が伸びているかどうかを判定します。`fingerIndices` 配列には 1 本の指の 4 つのキーポイント番号が含まれており、インデックス 3 が指先、インデックス 1 が第 2 関節に対応します。画面座標系では Y 軸が下向きのため、指先の Y 座標が関節より小さい（画面上方にある）場合に「伸びている」と判定します。

`checkDrawingMode()` 関数は、人差し指だけが伸びている状態を検出します。人差し指（5-8 番）と中指（9-12 番）の状態を確認し、人差し指が伸びていて中指が曲がっていれば描画モードとします。これにより、手を開いたり閉じたりするだけで描画の ON/OFF を直感的に切り替えられます。

`draw()` 関数では、描画モードの場合のみ人差し指の指先（8 番）の座標を取得し、`lineTo()` で線を描きます。前のフレームの位置から現在位置まで線が引かれるため、指を動かすと軌跡が描画されます。

```
1 function isFingerExtended(keypoints, fingerIndices) {
2     const tip = keypoints[fingerIndices[3]];  // 指先
3     const pip = keypoints[fingerIndices[1]];  // 第2関節
4     return tip.y < pip.y; // 指先が関節より上なら伸びている
5 }
6
7 function checkDrawingMode(hand) {
```

```
    const indexExtended = isFingerExtended(hand.keypoints, [5,6,7,8]);
    const middleExtended = isFingerExtended(hand.keypoints,
        [9,10,11,12]);
    // 人差し指だけ伸びていれば描画モード
    return indexExtended && !middleExtended;
}

function draw(hand) {
    if (checkDrawingMode(hand)) {
        const tip = hand.keypoints[8]; // 人差し指の指先
        drawCtx.lineTo(tip.x, tip.y);
        drawCtx.stroke();
    }
}
```

拡張のヒント

拡張のヒント：ジェスチャーで色・太さ変更

2 本指で太さ変更、3 本指で色変更など、ジェスチャーに機能を割り当てられます。

2.3 サンプル 13：フェイスフィルター

2.3.1 概要

SNS で人気の顔フィルター機能を実装します。サングラス、猫耳、マスクなど複数のフィルターを顔に適用し、ボタンで切り替えられます。顔のサイズや向きに応じてフィルターが自動調整されます。

2.3.2 学べること

- **複数フィルターの管理** – オブジェクトでフィルター情報を管理
- **顔の向きに応じた調整** – 目の位置から傾きを計算
- **動的な配置とスケーリング** – 顔のサイズに合わせてフィルターを拡大縮小

2.3.3 主要なコード

以下のコードは、複数のフィルターをオブジェクトで管理し、顔の位置と向きに合わせて描画する仕組みを示しています。

`filters` オブジェクトでは、各フィルターを名前をキーとして管理し、それぞれに `draw` 関数を持たせています。この設計により、フィルターの追加や切り替えが容易になります。

サングラスフィルターでは、まず左目（33 番）と右目（263 番）の位置から幅と傾きを計算します。`Math.atan2()` で目の傾きを角度として取得し、Canvas API の `save()`/`restore()` で描画状

態を保存・復元しながら、translate() で目の中心に移動、rotate() で傾きを適用してから絵文字を描画します。これにより、顔が傾いてもサングラスが自然に追従します。

猫耳フィルターは、額（10 番）の位置を基準に左右にオフセットして配置するシンプルな実装です。実際のアプリケーションでは顔のサイズに応じてオフセット量を調整します。

```
const filters = {
    sunglasses: {
        draw: (kp) => {
            const leftEye = kp[33], rightEye = kp[263];
            const width = distance(leftEye, rightEye) * 2;
            const angle = Math.atan2(rightEye.y - leftEye.y, rightEye.x -
                leftEye.x);
            ctx.save();
            ctx.translate((leftEye.x + rightEye.x)/2, (leftEye.y +
               rightEye.y)/2);
            ctx.rotate(angle);
            ctx.fillText(' 🆚', -width/2, 0);
            ctx.restore();
        }
    },
    catEars: {
        draw: (kp) => {
            const forehead = kp[10];
            ctx.fillText(' ', forehead.x - 40, forehead.y - 50);
            ctx.fillText(' ', forehead.x + 40, forehead.y - 50);
        }
    }
};
```

2.4 サンプル 14：Neural Network - 色分類

2.4.1 概要

このサンプルでは、ml5.js のニューラルネットワーク機能を使って、自分でモデルを訓練します。RGB スライダーで色を選択し、「暖色」「寒色」「中間色」などのカテゴリに分類するモデルを作成します。

機械学習の「訓練」と「推論」の違いを体験できる重要なサンプルです。

2.4.2 学べること

- **ml5.neuralNetwork() の初期化** – 入力、出力、タスクタイプの設定
- **訓練データの追加** – addData() で入出力ペアを登録
- **モデルの訓練** – train() でエポック数を指定して学習
- **推論の実行** – classify() で新しいデータを分類

2.4.3 主要なコード

以下のコードは、ニューラルネットワークの作成から訓練、推論までの一連の流れを示しています。

`ml5.neuralNetwork()` でニューラルネットワークを作成します。`inputs` で入力特徴量の名前（RGB 値）を、`outputs` で出力の名前を指定します。`task: 'classification'` は分類タスクであることを示し、`debug: true` を設定すると訓練中の損失値がコンソールに表示されます。

`addData()` で訓練データを追加します。第 1 引数が入力（RGB 値）、第 2 引数が正解ラベル（カテゴリ）です。機械学習では多くの訓練データが必要なため、各カテゴリに十分なサンプルを用意することが重要です。

`normalizeData()` はデータの正規化を行います。RGB 値は 0〜255 の範囲ですが、ニューラルネットワークは 0〜1 程度の値で最も効率的に学習するため、この前処理が必要です。

`train()` で訓練を開始します。`epochs` は訓練データ全体を何回繰り返し学習するかを指定します。訓練完了後、`classify()` で新しい入力に対する予測を取得できます。

```
// ニューラルネットワークの初期化
const options = {
    inputs: ['r', 'g', 'b'],
    outputs: ['category'],
    task: 'classification',
    debug: true  // 訓練状況を表示
};
const nn = ml5.neuralNetwork(options);

// 訓練データの追加
nn.addData({ r: 255, g: 0, b: 0 }, { category: 'warm' });
nn.addData({ r: 0, g: 0, b: 255 }, { category: 'cool' });
nn.addData({ r: 0, g: 255, b: 0 }, { category: 'neutral' });
// ... 多くのサンプルを追加

// 訓練の実行
nn.normalizeData();
nn.train({ epochs: 50 }, finishedTraining);

// 推論
async function classify(r, g, b) {
    const results = await nn.classify({ r, g, b });
    console.log(results[0].label, results[0].confidence);
}
```

実装のポイント

訓練データの重要性

モデルの精度は訓練データの質と量に大きく依存します。各カテゴリに偏りなく、十分な数の

サンプルを用意することが重要です。

拡張のヒント

拡張のヒント：モデルの保存と読み込み

```
nn.save('color-model');  // モデルをダウンロード保存
nn.load('color-model/model.json', modelLoaded);  // 読み込み
```

2.5 サンプル 15：住宅価格予測（回帰）

2.5.1 概要

サンプル 14 では「分類」を学びましたが、このサンプルでは「回帰」を学びます。面積と築年数から住宅価格を予測するモデルを訓練します。回帰は、カテゴリではなく連続的な数値を予測するタスクです。

2.5.2 学べること

- **回帰タスクの設定** – task: 'regression' を指定
- **データの正規化** – normalizeData() の重要性
- **predict() による数値予測** – classify() との違い

2.5.3 主要なコード

以下のコードは、回帰タスク（連続値の予測）における設定と使い方を示しています。

回帰タスクでは `task: 'regression'` を指定します。分類タスクが「どのカテゴリに属するか」を予測するのに対し、回帰タスクは「具体的な数値はいくつか」を予測します。住宅価格、気温、売上予測など、数値を出力する場面で使用します。

訓練データは分類と同様に `addData()` で追加しますが、出力も数値として指定します。入力（面積、築年数）と出力（価格）の関係をモデルが学習します。

回帰タスクでは `normalizeData()` が特に重要です。面積（60〜100）、築年数（5〜20）、価格（2800〜3500）のようにスケールが異なるデータを、モデルが扱いやすい範囲に統一します。正規化を忘れると学習がうまく進まないことがあります。

予測時は `classify()` ではなく `predict()` を使用します。結果には予測された数値が含まれ、`results[0].price` で価格にアクセスできます。

```
const options = {
    inputs: ['area', 'age'],
    outputs: ['price'],
    task: 'regression'  // 回帰タスク
```

```
};
const nn = ml5.neuralNetwork(options);

// 訓練データ（面積m2, 築年数, 価格万円）
nn.addData({ area: 60, age: 5 }, { price: 3000 });
nn.addData({ area: 80, age: 10 }, { price: 3500 });
nn.addData({ area: 100, age: 20 }, { price: 2800 });

nn.normalizeData();  // 正規化は必須
nn.train({ epochs: 100 }, finishedTraining);

// 予測
async function predict(area, age) {
    const results = await nn.predict({ area, age });
    console.log(`予測価格: ${results[0].price.toFixed(0)}万円`);
}
```

2.6 サンプル 16：体で音楽を奏でよう

2.6.1 概要

BodyPose と Web Audio API を組み合わせて、体の動きで音楽を演奏します。右手の X 座標で周波数（音程）を、Y 座標で音量を制御します。テルミンのような電子楽器を体験できます。

2.6.2 学べること

- **Web Audio API の基礎** – AudioContext, OscillatorNode, GainNode
- **座標から音響パラメータへの変換** – 線形/対数スケーリング
- **視覚的フィードバック** – 音に合わせた視覚エフェクト

2.6.3 主要なコード

以下のコードは、Web Audio API の基本構造と、体の位置を音響パラメータに変換する仕組みを示しています。

Web Audio API では、まず `AudioContext` を作成します。これが音声処理のメインコンテナです。`createOscillator()` で音を生成するオシレーター（発振器）を、`createGain()` で音量を制御するゲインノードを作成します。

注意

ブラウザの自動再生制限

多くのブラウザでは、ユーザー操作なしに音声を再生できません。`AudioContext` は初期状態

で suspended（停止中）になっていることがあります。「開始」ボタンのクリックイベント内で audioCtx.resume() を呼び出すことで、音声再生を有効化してください。

オーディオノードは connect() でチェーン接続します。オシレーター→ゲインノード→出力先（destination）の順に接続することで、音量調整された音が再生されます。oscillator.start() で発音を開始します。

updateSound() 関数では、右手首の座標を音響パラメータに変換します。X 座標（0〜canvas 幅）を周波数（200〜800Hz）にマッピングし、Y 座標（0〜canvas 高さ）を音量（0〜1）にマッピングします。setTargetAtTime() は値を滑らかに変化させる関数で、第 3 引数の時定数（0.1 秒）で変化の滑らかさを調整できます。

```
const audioCtx = new AudioContext();
const oscillator = audioCtx.createOscillator();
const gainNode = audioCtx.createGain();

oscillator.connect(gainNode);
gainNode.connect(audioCtx.destination);
oscillator.start();

function updateSound(keypoints) {
    const rightWrist = keypoints[10];

    // X座標 → 周波数（200Hz〜800Hz）
    const freq = 200 + (rightWrist.x / canvas.width) * 600;
    oscillator.frequency.setTargetAtTime(freq, audioCtx.currentTime,
        0.1);

    // Y座標 → 音量（上が大きい）
    const volume = 1 - (rightWrist.y / canvas.height);
    gainNode.gain.setTargetAtTime(volume, audioCtx.currentTime, 0.1);
}
```

2.7 サンプル 17：ジェスチャー認識

2.7.1 概要

HandPose を使って、グー、チョキ、パー、サムズアップ、ピースの 5 種類のジェスチャーを認識します。各指の状態（伸びている/曲がっている）をパターンマッチングして判定します。

2.7.2 学べること

- **指の状態判定ロジック** – 5 本の指それぞれの伸び/曲がりを判定
- **パターンマッチング** – 指の状態の組み合わせでジェスチャーを特定

- **安定化処理** – ノイズを除去するためのフィルタリング

2.7.3 主要なコード

以下のコードは、5 本の指の状態を取得し、パターンマッチングでジェスチャーを識別する仕組みを示しています。

`getFingerStates()` 関数は、5 本の指それぞれについて「伸びているかどうか」を真偽値の配列として返します。`fingers` 配列には各指のキーポイント番号が定義されており、`map()` で各指について `isFingerExtended()` を呼び出し、結果を配列にまとめます。

`recognizeGesture()` 関数では、5 本の指の状態を分割代入で個別の変数に取り出します。`[thumb, index, middle, ring, pinky]` という形式で、配列の各要素に名前を付けて受け取ります。

ジェスチャーの判定は、指の状態の組み合わせを if 文で順番にチェックします。例えば「グー」は 4 本指（人差し〜小指）がすべて曲がっている状態、「パー」はすべて伸びている状態です。「チョキ」と「ピース」は人差し指と中指が伸びている点で同じですが、親指の状態で区別します（親指が伸びていればピース、畳んでいればチョキ）。「サムズアップ」は親指のみ伸びている状態として定義しています。どのパターンにも一致しない場合は'unknown' を返します。

ヒント

条件の順序に注意

if 文の判定順序は重要です。より具体的な条件（例：サムズアップ）を先に判定し、一般的な条件を後にすることで、正しく分類できます。また、同じ条件が重複しないよう注意しましょう。

```
function getFingerStates(hand) {
    const fingers = [
        [1,2,3,4],    // 親指
        [5,6,7,8],    // 人差し指
        [9,10,11,12], // 中指
        [13,14,15,16],// 薬指
        [17,18,19,20] // 小指
    ];
    return fingers.map(f => isFingerExtended(hand.keypoints, f));
}

function recognizeGesture(hand) {
    const [thumb, index, middle, ring, pinky] = getFingerStates(hand);

    // 具体的な条件から順に判定
    if (thumb && !index && !middle && !ring && !pinky) return 'サムズアップ';
    if (!index && !middle && !ring && !pinky) return 'グー';
    if (index && middle && ring && pinky) return 'パー';
```

```
19      // チョキとピースは親指の状態で区別
20      if (index && middle && !ring && !pinky) {
21          return thumb ? 'ピース' : 'チョキ';
22      }
23      return 'unknown';
24  }
```

2.8 サンプル 18：マルチモデル同時使用

2.8.1 概要

BodyPose、HandPose、FaceMesh の 3 つのモデルを同時に実行し、それぞれの検出結果を 1 つの画面に統合表示します。各モデルはチェックボックスで個別に ON/OFF 切り替えができます。

2.8.2 学べること

- **複数モデルの並列初期化** – Promise.all() で効率的に初期化
- **パフォーマンス管理** – 不要なモデルを停止してリソースを節約
- **検出結果の統合描画** – 異なる色やスタイルで区別

2.8.3 主要なコード

以下のコードは、複数のモデルを効率的に初期化し、状態に応じて選択的に実行する仕組みを示しています。

`Promise.all()` を使うと、複数の非同期処理を並列に実行できます。3 つのモデルの初期化を順番に（直列に）行うと合計時間がかかりますが、並列に行うことで最も遅いモデルの読み込み時間だけで済みます。結果は配列で返され、分割代入で個別の変数に取り出しています。

`modelState` オブジェクトは、各モデルの ON/OFF 状態を管理します。UI のチェックボックスと連動させることで、ユーザーが必要なモデルだけを有効化できます。

`detect()` 関数では、`modelState` の状態を確認してから各モデルの検出を実行します。不要なモデルの処理をスキップすることで、CPU リソースを節約し、フレームレートを向上させることができます。各モデルの検出結果は異なる色で描画することで、どのモデルによる検出かを視覚的に区別できます。

```
1  // 並列初期化
2  const [bodyPose, handPose, faceMesh] = await Promise.all([
3      ml5.bodyPose('MoveNet'),
4      ml5.handPose(),
5      ml5.faceMesh()
6  ]);
7
8  const modelState = { body: true, hand: true, face: true };
```

```
9
10 async function detect() {
11     if (modelState.body) {
12         const bodies = await bodyPose.detect(video);
13         drawBodies(bodies, '#FF6B6B');
14     }
15     if (modelState.hand) {
16         const hands = await handPose.detect(video);
17         drawHands(hands, '#4ECDC4');
18     }
19     if (modelState.face) {
20         const faces = await faceMesh.detect(video);
21         drawFaces(faces, '#45B7D1');
22     }
23     requestAnimationFrame(detect);
24 }
```

2.9 サンプル 19：音声ビジュアライザー + 分類

2.9.1 概要

マイク入力の周波数をリアルタイムで可視化しながら、音声コマンドを分類します。Web Audio API の AnalyserNode を使って周波数データを取得し、円形のビジュアライザーとして描画します。

2.9.2 学べること

- **AnalyserNode の使用** – 周波数データの取得
- **getByteFrequencyData()** – 周波数帯域ごとの音量を配列で取得
- **ビジュアライザーの描画** – 円形、棒グラフなどの可視化手法

2.9.3 主要なコード

以下のコードは、Web Audio API の AnalyserNode を使った周波数解析と円形ビジュアライザーの描画を示しています。

`createAnalyser()` で AnalyserNode を作成します。`fftSize` は FFT（高速フーリエ変換）のサイズで、大きいほど周波数の分解能が高くなりますが処理負荷も増えます。256 は一般的な値です。`frequencyBinCount` は `fftSize / 2` で、周波数バンドの数を表します。この配列に周波数データが格納されます。

`drawVisualizer()` 関数では、まず `getByteFrequencyData()` で現在の周波数データを取得します。各要素は 0〜255 の範囲で、その周波数帯域の音量を表します。

円形ビジュアライザーは、中心点から各周波数バンドに対応する方向に棒を描画します。角度は `(i / dataArray.length) * Math.PI * 2` で 360 度を均等に分割し、`Math.cos()` と `Math.sin()` で

始点・終点の座標を計算します。棒の長さは周波数データの値に比例させています。

```
const analyser = audioCtx.createAnalyser();
analyser.fftSize = 256;
const dataArray = new Uint8Array(analyser.frequencyBinCount);

function drawVisualizer() {
    analyser.getByteFrequencyData(dataArray);

    const centerX = canvas.width / 2;
    const centerY = canvas.height / 2;
    const radius = 100;

    dataArray.forEach((value, i) => {
        const angle = (i / dataArray.length) * Math.PI * 2;
        const barHeight = value / 2;
        const x1 = centerX + Math.cos(angle) * radius;
        const y1 = centerY + Math.sin(angle) * radius;
        const x2 = centerX + Math.cos(angle) * (radius + barHeight);
        const y2 = centerY + Math.sin(angle) * (radius + barHeight);

        ctx.beginPath();
        ctx.moveTo(x1, y1);
        ctx.lineTo(x2, y2);
        ctx.stroke();
    });
}
```

2.10 サンプル 20：Image Classifier - カスタム UI

2.10.1 概要

実用的な UI を備えた画像分類アプリケーションです。ライブモード（カメラ映像をリアルタイム分類）とキャプチャモード（静止画を撮影して分類）を切り替えられ、分類履歴も表示されます。

2.10.2 学べること

- **モード管理** – 状態に応じた UI 切り替え
- **キャプチャ機能** – canvas.toDataURL() で画像を保存
- **履歴管理** – 配列で結果を蓄積し、リスト表示

2.10.3 主要なコード

以下のコードは、モード切り替え、画像キャプチャ、履歴管理の実装パターンを示しています。

mode 変数でアプリケーションの状態を管理します。'live' モードではリアルタイム分類、'capture' モードでは静止画を撮影して分類します。UI ボタンでモードを切り替えることで、異なる機能を提供できます。

captureImage() 関数では、ctx.drawImage() でビデオフレームを Canvas に描画し、canvas.toDataURL() で Base64 エンコードされた画像データを取得します。この形式は<img>タグの src に直接設定でき、サーバーへの送信も可能です。

classifyAndSave() 関数では、キャプチャと分類を連続して行い、結果を履歴配列に保存します。unshift() は配列の先頭に要素を追加するメソッドで、新しい結果が常に一番上に表示されます。履歴には画像データ、分類結果、タイムスタンプを含むオブジェクトを保存し、後で一覧表示や再確認ができるようにしています。

```
let mode = 'live'; // 'live' or 'capture'
const history = [];

function captureImage() {
    ctx.drawImage(video, 0, 0);
    const imageData = canvas.toDataURL('image/jpeg');
    return imageData;
}

async function classifyAndSave() {
    const imageData = captureImage();
    const results = await classifier.classify(canvas);

    history.unshift({
        image: imageData,
        label: results[0].label,
        confidence: results[0].confidence,
        time: new Date().toLocaleTimeString()
    });

    updateHistoryDisplay();
}
```

第 3 章

上級編：実践的なアプリケーション

この章の到達目標と所要時間

到達目標

- 独自のポーズやジェスチャーを認識するカスタム分類器を構築できる
- 複数人の同時追跡やマルチモーダル入力を扱うシステムを設計できる
- 実用的なアプリケーション（フィットネス、教育、エンタメ）を開発できる

推奨所要時間

- 全サンプル通し：90 分 ×5〜6 回
- 重点サンプル（21, 25, 28, 30）のみ：90 分 ×3 回

この章では、これまで学んだ知識を総合して、より高度で実用的なアプリケーションを作成します。カスタムモデルの訓練、複数人の追跡、複合的なシステム構築など、実際のプロダクト開発に直結するスキルを習得します。

3.1　サンプル 21：カスタムポーズ分類器

3.1.1　概要

BodyPose で検出したキーポイントデータを使って、ユーザー定義のポーズを分類するカスタムモデルを訓練します。「バンザイ」「T ポーズ」「片手を上げる」など、任意のポーズを登録して認識させることができます。

3.1.2　学べること

- **ポーズデータの前処理** – 17 キーポイント ×2 座標=34 次元の入力
- **座標の正規化** – 体の位置やサイズに依存しない特徴量
- **リアルタイム訓練と推論** – ブラウザ上で完結

3.1.3 主要なコード

以下のコードは、BodyPose のキーポイントデータをニューラルネットワークの入力に変換し、カスタムポーズを学習させる仕組みを示しています。

ニューラルネットワークの入力サイズは 34 次元（17 キーポイント ×2 座標）です。各キーポイントの x、y 座標を平坦化した配列として渡します。これにより、体の姿勢全体を数値ベクトルとして表現できます。

`poseToInput()` 関数は、キーポイントを正規化された入力ベクトルに変換します。正規化は機械学習において非常に重要で、ここでは鼻の位置を原点（0, 0）として、肩幅を基準にスケーリングしています。これにより、人物の位置（画面上のどこにいるか）やカメラからの距離（体の大きさ）に依存しない特徴量を得られます。同じポーズであれば、画面の左にいても右にいても、同じような入力値になります。

`recordPose()` 関数では、現在のキーポイントを正規化して入力とし、ユーザーが指定したラベルと組み合わせて訓練データに追加します。この関数を各ポーズで複数回呼び出すことで、訓練データセットを構築します。

```
const nn = ml5.neuralNetwork({
    inputs: 34,  // 17 keypoints * 2 (x, y)
    outputs: ['pose'],
    task: 'classification'
});

function poseToInput(keypoints) {
    const input = [];
    // 正規化: 鼻を原点、肩幅を基準にスケーリング
    const nose = keypoints[0];
    const shoulderWidth = distance(keypoints[5], keypoints[6]);

    keypoints.forEach(kp => {
        input.push((kp.x - nose.x) / shoulderWidth);
        input.push((kp.y - nose.y) / shoulderWidth);
    });
    return input;
}

function recordPose(label) {
    const input = poseToInput(currentKeypoints);
    nn.addData(input, { pose: label });
}
```

3.2 サンプル 22：フィットネストラッカー

3.2.1 概要

スクワット、腕立て伏せ、ランジ、ジャンピングジャックの 4 種類のエクササイズを自動判定し、回数をカウントする本格的なフィットネスアプリです。フォームの評価やワークアウトログの記録機能も備えています。

3.2.2 学べること

- **複数エクササイズの状態管理** – 各エクササイズの判定ロジック
- **フォーム評価** – 理想的な角度との比較でフィードバック
- **ワークアウトログ** – セッションの記録と統計

3.2.3 主要なコード

以下のコードは、複数のエクササイズを汎用的に管理し、同じロジックで異なる運動をカウントする設計パターンを示しています。

`exercises` オブジェクトは、各エクササイズの設定をまとめて管理します。`checkJoints` は角度を計算する 3 つの関節のキーポイント番号、`downThreshold` と `upThreshold` は動作の下限・上限角度、`state` は現在の状態、`count` は回数です。この構造により、新しいエクササイズを追加する際はオブジェクトに項目を追加するだけで済みます。

`updateExercise()` 関数は、エクササイズ名とキーポイントを受け取り、汎用的にカウント処理を行います。まず指定された 3 関節の角度を計算し、閾値と現在の状態に基づいて状態遷移とカウントアップを判定します。この関数を各エクササイズで呼び出すことで、コードの重複を避けながら複数の運動をサポートできます。

スクワットは膝の角度（腰-膝-足首）、腕立て伏せは肘の角度（肩-肘-手首）を監視するという違いがありますが、同じ関数で処理できるのがこの設計の利点です。

```
const exercises = {
    squat: {
        checkJoints: [12, 14, 16], // 右腰、右膝、右足首
        downThreshold: 90,
        upThreshold: 160,
        state: 'up',
        count: 0
    },
    pushup: {
        checkJoints: [6, 8, 10], // 右肩、右肘、右手首
        downThreshold: 90,
        upThreshold: 160,
```

```
        state: 'up',
        count: 0
    }
};

function updateExercise(name, keypoints) {
    const ex = exercises[name];
    const angle = calculateAngle(
        keypoints[ex.checkJoints[0]],
        keypoints[ex.checkJoints[1]],
        keypoints[ex.checkJoints[2]]
    );

    if (angle < ex.downThreshold && ex.state === 'up') {
        ex.state = 'down';
    } else if (angle > ex.upThreshold && ex.state === 'down') {
        ex.state = 'up';
        ex.count++;
    }
}
```

3.3 サンプル 23：バーチャルキーボード

3.3.1 概要

画面上に仮想キーボードを表示し、手指でタップして文字を入力します。指先とキーの当たり判定、タップ検出（指を押し込む動作)、デバウンス処理など、実用的な入力システムを実装します。

3.3.2 学べること

- **キーボードレイアウトの生成** – 動的にキーを配置
- **タップ検出** – 指先の Z 座標変化または位置変化で判定
- **デバウンス処理** – 連続入力を防止

3.3.3 主要なコード

以下のコードは、キーボードレイアウトの定義、タップ検出、当たり判定の実装を示しています。

`keyboard` 配列は、QWERTY レイアウトを 2 次元配列で表現しています。各行がキーボードの 1 段に対応し、配列のインデックスから画面上の位置を計算できます。この構造により、レイアウトの変更や多言語対応も容易に行えます。

`checkKeyPress()` 関数は、指の縦方向の動きからタップを検出します。現在の指先 Y 座標と前フレームの Y 座標を比較し、差が閾値を超えていればタップと判定します。この単純なアルゴリズム

により、指を下に押し込む動作を検出できます。実際のアプリケーションでは、デバウンス処理（連続タップの防止）も必要です。

`getKeyAtPosition()` 関数は、指先の座標からどのキーが押されたかを特定します。キーボードの描画位置と同じ計算式を使い、座標がどのキーの範囲内にあるかを判定します。各行は半キー分ずつずれている（QWERTY 配列の特徴）ため、`rowIndex * keyWidth / 2` でオフセットを加えています。

注意

forEach と return の注意点

`forEach` のコールバック内で `return` しても、外側の関数からは戻れません。値を返したい場合は `for` ループを使用します。これは JavaScript でよくある落とし穴です。

```
const keyboard = [
    ['Q','W','E','R','T','Y','U','I','O','P'],
    ['A','S','D','F','G','H','J','K','L'],
    ['Z','X','C','V','B','N','M']
];

function checkKeyPress(fingerTip, lastY) {
    // 指が下に動いたらタップと判定
    const tapThreshold = 20;
    return fingerTip.y - lastY > tapThreshold;
}

function getKeyAtPosition(x, y) {
    const keyWidth = canvas.width / 10;
    const keyHeight = 60;
    const startY = canvas.height - 200;

    // forループを使用（forEachではreturnが効かない）
    for (let rowIndex = 0; rowIndex < keyboard.length; rowIndex++) {
        const row = keyboard[rowIndex];
        for (let keyIndex = 0; keyIndex < row.length; keyIndex++) {
            const keyX = keyIndex * keyWidth + (rowIndex * keyWidth / 2);
            const keyY = startY + rowIndex * keyHeight;
            if (x > keyX && x < keyX + keyWidth &&
                y > keyY && y < keyY + keyHeight) {
                return row[keyIndex];
            }
        }
    }
    return null;
}
```

3.4 サンプル 24：頭の向き推定

3.4.1 概要

FaceMesh の 2D キーポイントから、顔の 3D 回転（Yaw: 左右、Pitch: 上下、Roll: 傾き）を推定します。注意度メーターや、3D キューブの回転で視覚的に表示します。

3.4.2 学べること

- **2D から 3D 姿勢の推定** – 顔の特徴点の相対位置から角度を計算
- **三角関数による角度計算** – Math.atan2 の活用
- **CSS 3D Transform による可視化**

3.4.3 主要なコード

以下のコードは、2D の顔キーポイントから 3D の頭部回転（Yaw、Pitch、Roll）を推定する手法を示しています。

`estimateHeadPose()` 関数では、顔の主要なランドマーク（鼻、両目、あご、額）を取得し、それらの相対的な位置関係から 3 軸の回転角度を計算します。

Yaw（左右の首振り）は、鼻の位置が両目の中心からどれだけずれているかで推定します。顔が右を向くと鼻は目の中心より右側に、左を向くと左側に移動します。このずれを目の間隔で正規化し、角度に変換しています。

Pitch（上下の首振り）は、顔の縦方向のバランスで推定します。顔の高さ（額からあご）に対する鼻の位置の比率を計算し、正面を向いているときの基準値（0.4）との差から角度を求めます。上を向くと鼻は相対的に下がり、下を向くと上がります。

Roll（首の傾き）は、両目の傾きから直接計算できます。`Math.atan2()` で目を結ぶ線の角度を求め、ラジアンから度に変換します。

```
function estimateHeadPose(keypoints) {
    const nose = keypoints[1];
    const leftEye = keypoints[33];
    const rightEye = keypoints[263];
    const chin = keypoints[152];
    const forehead = keypoints[10];

    // Yaw: 鼻と目の中心のずれ
    const eyeCenter = {
        x: (leftEye.x + rightEye.x) / 2,
        y: (leftEye.y + rightEye.y) / 2
    };
    const yaw = (nose.x - eyeCenter.x) / distance(leftEye, rightEye) *
        90;
```

```
14
15      // Pitch: 鼻と額・顎の関係
16      const faceHeight = distance(forehead, chin);
17      const noseRatio = (nose.y - forehead.y) / faceHeight;
18      const pitch = (noseRatio - 0.4) * 180;
19
20      // Roll: 目の傾き
21      const roll = Math.atan2(rightEye.y - leftEye.y, rightEye.x - leftEye.
            x) * 180 / Math.PI;
22
23      return { yaw, pitch, roll };
24  }
```

3.5 サンプル 25：マルチパーソントラッキング

3.5.1 概要

最大 6 人の姿勢を同時に追跡し、各人物に ID を割り当てて個別に統計情報（移動距離、速度など）を表示します。フレーム間の人物マッチングがキーポイントです。

3.5.2 学べること

- **maxPoses オプション** – 複数人検出の設定
- **距離ベースのマッチング** – フレーム間で同一人物を追跡
- **個人ごとの履歴管理** – ID に紐づいた情報の蓄積

3.5.3 主要なコード

以下のコードは、複数人を検出し、フレーム間で同一人物を追跡するマッチングアルゴリズムを示しています。

`maxPoses: 6` オプションにより、最大 6 人まで同時に検出できます。デフォルトは 1 人のみの検出のため、複数人を扱う場合はこのオプションが必要です。

`persons` は Map オブジェクトで、各人物の ID をキーとしてデータ（位置履歴、移動距離など）を管理します。Map は追加・削除が頻繁に発生するデータに適しています。

`matchPersons()` 関数は、現フレームで検出されたポーズと、前フレームで追跡していた人物をマッチングします。各ポーズの中心点（キーポイントの重心）を計算し、既存の人物との距離を比較します。最も近い人物が閾値（100 ピクセル）以内であればその人物と判定し、`updatePerson()` で情報を更新します。どの既存人物とも一致しない場合は新しい人物として `createNewPerson()` で追加します。

このシンプルな距離ベースマッチングは、人物が重なったり一時的に検出されなくなったりする場合に誤りが生じる可能性がありますが、多くの場面で実用的に機能します。

```
const bodyPose = await ml5.bodyPose('MoveNet', { maxPoses: 6 });
const persons = new Map(); // ID -> person data

function matchPersons(poses) {
    poses.forEach(pose => {
        const center = getCenter(pose.keypoints);
        let matchedId = null;
        let minDist = 100; // マッチング閾値

        persons.forEach((person, id) => {
            const dist = distance(center, person.lastPosition);
            if (dist < minDist) {
                minDist = dist;
                matchedId = id;
            }
        });

        if (matchedId) {
            updatePerson(matchedId, pose);
        } else {
            createNewPerson(pose);
        }
    });
}
```

3.6 サンプル 26：時系列予測

3.6.1 概要

過去 10 時点のデータから次の値を予測するニューラルネットワークを訓練します。サイン波、ランダムウォーク、トレンドなど 4 種類のパターンで学習し、予測精度を比較できます。

3.6.2 学べること

- **シーケンシャルデータの入力形式** – 過去 N 個の値を入力
- **カスタムレイヤー構造** – layers オプションでネットワーク構造を定義
- **予測と実測の比較** – グラフで精度を可視化

3.6.3 主要なコード

以下のコードは、時系列データを入力として次の値を予測するニューラルネットワークの構築方法を示しています。

SEQUENCE_LENGTH は入力として使用する過去の時点数です。10 を指定すると、過去 10 個の値か

ら次の 1 個を予測します。この数値はデータの特性や予測精度に影響するため、実験的に調整します。

ニューラルネットワークの layers オプションで、カスタムのネットワーク構造を定義できます。各層は type（層の種類）、units（ニューロン数）、activation（活性化関数）で構成されます。'dense' は全結合層、'relu' は負の値を 0 にする活性化関数、'linear' は出力をそのまま返す関数です。入力から出力に向かってユニット数を減らしていく構造は、情報を圧縮しながら特徴を抽出する一般的なパターンです。

prepareData() 関数は、連続した時系列データからスライディングウィンドウで訓練データを生成します。例えば 10 個のデータ [0,1,2,...,9] がある場合、最初の訓練例は入力 [0-9]、出力 [10] となります。slice() でウィンドウを切り出し、その直後の値を正解として addData() します。

```
const SEQUENCE_LENGTH = 10;

const nn = ml5.neuralNetwork({
    inputs: SEQUENCE_LENGTH,
    outputs: 1,
    task: 'regression',
    layers: [
        { type: 'dense', units: 32, activation: 'relu' },
        { type: 'dense', units: 16, activation: 'relu' },
        { type: 'dense', units: 1, activation: 'linear' }
    ]
});

function prepareData(series) {
    for (let i = SEQUENCE_LENGTH; i < series.length; i++) {
        const input = series.slice(i - SEQUENCE_LENGTH, i);
        const output = [series[i]];
        nn.addData(input, output);
    }
}
```

3.7 サンプル 27：ポーズマッチングゲーム

3.7.1 概要

画面に表示されるポーズを制限時間内に真似するゲームです。ポーズの一致度をスコアリングし、連続成功でコンボボーナスが加算されます。

3.7.2 学べること

- **ポーズの定義と判定** – 複数の条件を組み合わせた判定
- **一致度のスコアリング** – 完全一致/部分一致の段階評価

- **ゲームフロー管理** – タイマー、ラウンド、コンボシステム

3.7.3 主要なコード

以下のコードは、複数のポーズをルールベースで定義し、判定する仕組みを示しています。

poses オブジェクトは、各ポーズを名前と判定関数のペアで定義します。判定関数はキーポイントを受け取り、そのポーズが成立しているかどうかを真偽値で返します。この設計により、新しいポーズを追加する際は判定関数を実装するだけで済みます。

「T ポーズ」の判定では、左腕（左肩-左肘-左手首）と右腕（右肩-右肘-右手首）の角度をそれぞれ計算し、両方が 160 度以上（ほぼ真っ直ぐ）であれば T ポーズと判定します。腕を水平に伸ばすと肘の角度は 180 度に近づくため、160 度という閾値で多少の曲がりを許容しています。

「バンザイ」の判定は、両手首の Y 座標が対応する肩の Y 座標より小さい（画面上で上にある）かどうかで判断します。角度を使わないシンプルな方法ですが、手を上げる動作を確実に検出できます。

checkPoseMatch() 関数は、ターゲットポーズの名前をキーにして判定関数を呼び出します。ゲームのメインループでは、表示中のお題ポーズとプレイヤーの姿勢を比較し、一致すればスコアを加算する形で使用します。

```
const poses = {
    'Tポーズ': {
        check: (kp) => {
            const leftArm = calculateAngle(kp[5], kp[7], kp[9]);
            const rightArm = calculateAngle(kp[6], kp[8], kp[10]);
            return leftArm > 160 && rightArm > 160;
        }
    },
    'バンザイ': {
        check: (kp) => {
            return kp[9].y < kp[5].y && kp[10].y < kp[6].y;
        }
    }
};

function checkPoseMatch(targetPose, keypoints) {
    return poses[targetPose].check(keypoints);
}
```

3.8 サンプル 28：指文字認識システム

3.8.1 概要

アルファベットの指文字 12 種類（A, B, C, D, E, F, I, L, O, V, W, Y）を認識し、文字入力を行います。検出履歴を使って誤認識を減らし、安定した入力を実現します。

3.8.2 学べること

- **複雑な指状態パターンの定義** – 各文字の指の形をルール化
- **検出履歴による安定化** – 複数フレームで一致した場合のみ確定
- **入力バッファ管理** – 文字列の編集機能

3.8.3 主要なコード

以下のコードは、指文字パターンの定義と、検出結果を安定化させるヒストリーベースの確認メカニズムを示しています。

`fingerSpelling` オブジェクトは、各アルファベットの指の形を記述します。`thumb` は親指の状態（'side' は横向き、'across' は手のひらを横切る、'curved' は曲げるなど）、`fingers` は人差し指から小指までの伸び/曲げ状態を配列で表します。例えば'L' は親指と人差し指だけを伸ばし、'V' は人差し指と中指だけを伸ばすパターンです。

リアルタイムの検出結果はノイズが多く、フレームごとに異なる文字が検出されることがあります。`detectionHistory` 配列は直近の検出結果を保持し、`confirmLetter()` 関数で安定化処理を行います。

`confirmLetter()` 関数は、新しい検出結果を履歴に追加し、最大 `CONFIRM_FRAMES` 個（10 フレーム）を保持します。履歴の中で同じ文字が 8 回以上出現していれば、その文字を確定として返します。これにより、一時的な誤検出を除去し、ユーザーが意図した文字のみを入力できます。

```
const fingerSpelling = {
    'A': { thumb: 'side', fingers: [false, false, false, false] },
    'B': { thumb: 'across', fingers: [true, true, true, true] },
    'C': { thumb: 'curved', fingers: 'curved' },
    'L': { thumb: true, fingers: [true, false, false, false] },
    'V': { thumb: false, fingers: [true, true, false, false] },
    'Y': { thumb: true, fingers: [false, false, false, true] }
};

const detectionHistory = [];
const CONFIRM_FRAMES = 10;

function confirmLetter(letter) {
    detectionHistory.push(letter);
    if (detectionHistory.length > CONFIRM_FRAMES) {
        detectionHistory.shift();
    }
    // 8/10フレーム以上一致で確定
    const count = detectionHistory.filter(l => l === letter).length;
    return count >= 8 ? letter : null;
}
```

3.9 サンプル 29：AI お絵かきアシスタント

3.9.1 概要

Canvas 上に絵を描いている間、AI がリアルタイムで「何を描いているか」を予測して表示します。描画が進むにつれて予測が変化していく様子を観察できます。

3.9.2 学べること

- **Canvas 描画と AI 分類の統合** – 描画イベントと分類のタイミング制御
- **DoodleNet モデル** – 手書きスケッチに特化したモデル
- **プログレッシブな予測** – 描画量に応じた分類頻度の調整

3.9.3 主要なコード

以下のコードは、描画イベントと分類タイミングの連携、および DoodleNet モデルの使用方法を示しています。

`ml5.imageClassifier('DoodleNet')` で DoodleNet モデルを読み込みます。DoodleNet は Google Quick, Draw|データセットで訓練されたモデルで、手書きのシンプルなスケッチを認識するのに特化しています。MobileNet が写真画像向けなのに対し、DoodleNet は線画やイラストの認識に適しています。

`strokeCount` 変数は描画されたストローク（線を引いてマウスを離すまでの一連の動作）の数を追跡します。`'mouseup'` イベントでストローク完了を検知し、カウントを増やします。

分類は 3 ストロークごとに実行します。毎ストローク分類すると処理負荷が高く、ユーザー体験も落ちます。かといって描画完了まで待つとリアルタイム性が失われます。3 という数値は実験的に決めた値で、早すぎず遅すぎない適度なタイミングです。

`displayPredictions()` 関数では上位 5 件の予測結果を表示します。描き始めは曖昧な予測が多くなりますが、描画が進むにつれて特定のカテゴリへの確信度が高まっていく様子を観察できます。

```
const classifier = await ml5.imageClassifier('DoodleNet');
let strokeCount = 0;

canvas.addEventListener('mouseup', async () => {
    strokeCount++;
    // 3ストロークごとに分類
    if (strokeCount % 3 === 0) {
        const results = await classifier.classify(canvas);
        displayPredictions(results.slice(0, 5));
    }
});

```

```
function displayPredictions(results) {
    results.forEach((r, i) => {
        console.log(`${i+1}. ${r.label}: ${(r.confidence*100).toFixed
            (1)}%`);
    });
}
```

3.10 サンプル 30：総合 AI インタラクティブ体験

3.10.1 概要

BodyPose、HandPose、FaceMesh のすべてのモデルを統合し、6 つのモードと 8 種類の視覚エフェクトを提供する総合デモです。本ガイドで学んだ技術の集大成です。

3.10.2 6 つのモード

1. **ボディモード**: 全身のスケルトン表示
2. **ハンドモード**: 手の詳細表示
3. **フェイスモード**: 顔のランドマーク表示
4. **複合モード**: 全モデル同時使用
5. **パーティクルモード**: 動きに連動したパーティクル
6. **ミラーモード**: アート効果付きミラー

3.10.3 学べること

- **大規模なモード管理** – 状態パターンによる整理
- **パーティクルシステム** – 物理シミュレーションの基礎
- **エフェクト切り替え** – 再利用可能なエフェクト関数

3.10.4 主要なコード

以下のコードは、モード管理、パーティクルシステム、エフェクト切り替えの実装パターンを示しています。

`modes` 配列と `effects` 配列で、利用可能なモードとエフェクトを定義します。配列で管理することで、UI のボタン生成やサイクル切り替え（次のモードへ進む）が容易になります。`currentMode` と `currentEffect` で現在の状態を追跡します。

`particles` 配列は画面上のパーティクル（粒子）を管理します。各パーティクルは位置（x, y）、速度（vx, vy）、寿命（life）、色を持つオブジェクトです。

`updateParticles()` 関数は、キーポイントの位置を基にパーティクルを生成・更新します。各キーポイントで 10% の確率（`Math.random() < 0.1`）で新しいパーティクルを生成し、ランダムな

方向に速度を与えます。既存のパーティクルは毎フレーム位置を更新し、寿命を減らします。寿命が 0 以下になったパーティクルは splice() で配列から削除します。

このシンプルなパーティクルシステムは、動きに連動した視覚エフェクト（火花、軌跡、オーラなど）を作成する基礎となります。寿命、速度、生成確率を調整することで様々な効果を表現できます。

注意

配列のループ中の削除

forEach でループしながら splice で要素を削除すると、インデックスがずれて要素をスキップすることがあります。削除を含むループは**後ろから走査**するのが安全です。

```
const modes = ['body', 'hand', 'face', 'combined', 'particle', 'mirror'];
const effects = ['normal', 'neon', 'rainbow', 'pixelate', 'blur', 'invert
    ', 'sketch', 'thermal'];
let currentMode = 'combined';
let currentEffect = 'normal';

const particles = [];

function updateParticles(keypoints) {
    // キーポイントからパーティクルを生成
    keypoints.forEach(kp => {
        if (Math.random() < 0.1) {
            particles.push({
                x: kp.x,
                y: kp.y,
                vx: (Math.random() - 0.5) * 5,
                vy: (Math.random() - 0.5) * 5,
                life: 60,
                color: getEffectColor()
            });
        }
    });

    // パーティクルの更新（後ろから走査して安全に削除）
    for (let i = particles.length - 1; i >= 0; i--) {
        const p = particles[i];
        p.x += p.vx;
        p.y += p.vy;
        p.life--;
        if (p.life <= 0) particles.splice(i, 1);
    }
}
```

拡張のヒント

拡張のヒント：録画・共有機能

MediaRecorder API を使って、パフォーマンスを録画し共有できます。

```
const stream = canvas.captureStream(30);
const recorder = new MediaRecorder(stream);
const chunks = [];
recorder.ondataavailable = e => chunks.push(e.data);
recorder.onstop = () => {
    const blob = new Blob(chunks, { type: 'video/webm' });
    const url = URL.createObjectURL(blob);
    // ダウンロードリンクを作成
};
```

付録 A

ml5.js 関数・メソッド一覧

A.1 ml5.imageClassifier()

```
const classifier = await ml5.imageClassifier(model, options);
```

パラメータ: model（'MobileNet', 'DarkNet', 'DoodleNet' 等）
メソッド: `classify(input, num)` – 入力を分類

A.2 ml5.bodyPose()

```
const bodyPose = await ml5.bodyPose(model, options);
```

パラメータ: model（'MoveNet', 'BlazePose'）、options（maxPoses, flipHorizontal 等）
メソッド: `detect(input)` – ポーズを検出

A.3 ml5.handPose()

```
const handPose = await ml5.handPose(options);
```

パラメータ: options（maxHands, flipHorizontal 等）
メソッド: `detect(input)` – 手を検出

A.4 ml5.faceMesh()

```
const faceMesh = await ml5.faceMesh(options);
```

パラメータ: options（maxFaces, refineLandmarks 等）
メソッド: `detect(input)` – 顔を検出

A.5 ml5.soundClassifier()

```
const classifier = await ml5.soundClassifier(model, options);
```

パラメータ: model（'SpeechCommands18w' 等）
メソッド: `classify(callback)` – 音声を分類

A.6 ml5.neuralNetwork()

```
const nn = ml5.neuralNetwork(options);
```

オプション: inputs, outputs, task（'classification' | 'regression'）, debug, layers
メソッド:

- `addData(input, output)` – 訓練データを追加
- `normalizeData()` – データを正規化
- `train(options, callback)` – モデルを訓練
- `classify(input)` – 分類
- `predict(input)` – 回帰予測
- `save(name)` – モデルを保存
- `load(path, callback)` – モデルを読み込み

付録 B

LLM プロンプト集 — ml5.js を使いこなすために

本章では、ChatGPT、Claude、Gemini などの大規模言語モデル（LLM）を活用して ml5.js の学習を効率化するためのプロンプト集を紹介します。これらのプロンプトは、本ガイドで学んだ内容を深め、実践的なプロジェクト開発を支援することを目的としています。

B.1　準備：LLM を活用した学習の心構え

LLM を ml5.js 学習のパートナーとして活用する前に、以下の点を理解しておきましょう。

ヒント

効果的なプロンプトの基本原則

1. **具体的に**: 曖昧な質問より、具体的な状況や目標を伝える
2. **段階的に**: 一度に多くを求めず、段階的に質問を深める
3. **コンテキストを共有**: 自分のスキルレベルや使用環境を明示する
4. **コードを提示**: エラーや問題がある場合は、実際のコードを含める
5. **検証する**: LLM の回答は必ず自分で検証・テストする

注意

LLM は最新の ml5.js v1 の仕様を完全に把握していない場合があります。公式ドキュメント（https://docs.ml5js.org/）との照合を忘れずに行いましょう。

B.1.1　プロンプトの基本構造

効果的なプロンプトは以下の要素を含みます：

```
[役割の指定]
あなたはml5.jsの専門家です。
```

```
3
4  [背景・コンテキスト]
5  私はJavaScript初心者で、ml5.jsを使った
6  画像分類アプリを作りたいと考えています。
7
8  [具体的な質問・依頼]
9  MobileNetを使った基本的な画像分類の
10 コードを教えてください。
11
12 [制約条件]
13 - 単一HTMLファイルで完結させてください
14 - コメントを日本語で付けてください
```

B.2 初級プロンプト集

以下のプロンプトは、ml5.js の基本を学ぶ初心者向けです。本ガイドの第 1 章（初心者編）の内容と連携しています。

B.2.1 プロンプト 1：画像分類の基礎理解

```
1  ml5.jsのimageClassifierとMobileNetについて
2  教えてください。以下の点を含めて説明を
3  お願いします：
4  - MobileNetとは何か
5  - なぜ事前学習済みモデルを使うのか
6  - 分類できる画像の種類と数
7  - 信頼度（confidence）の意味
```

ガイドとの関連: サンプル 01「画像分類基本」、サンプル 02「Web カメラ画像分類」で学ぶ MobileNet の理論的背景を深く理解できます。

B.2.2 プロンプト 2：非同期処理の理解

```
1  JavaScriptのasync/awaitについて、
2  ml5.jsでの使い方を中心に教えてください。
3
4  以下のコードがなぜこの順序で書かれているのか
5  説明してください：
6
7  const classifier = await ml5.imageClassifier(
8    'MobileNet'
9  );
10 const results = await classifier.classify(img);
11
```

```
なぜawaitが必要なのですか？
awaitを書かないとどうなりますか？
```

ガイドとの関連: 全サンプルで使用されている async/await 構文の理解を深めます。

B.2.3 プロンプト 3：bodyPose のキーポイント理解

```
ml5.jsのbodyPose（MoveNet）が検出する
17個のキーポイントについて、
それぞれの名前と位置を教えてください。

また、各キーポイントのインデックス番号と
対応する身体部位のリストを作成してください。
```

ガイドとの関連: サンプル 03「ポーズ検出基本」、サンプル 04「スケルトン描画」で学ぶ bodyPose の詳細理解に役立ちます。

B.2.4 プロンプト 4：Canvas への描画基礎

```
HTML5 Canvasを使って、ml5.jsのbodyPoseで
検出したキーポイントを描画する方法を
教えてください。

以下を含めてください：
- 円でキーポイントを表示する方法
- 線でスケルトンを描く方法
- Webカメラ映像を左右反転して表示する方法
```

ガイドとの関連: サンプル 04「スケルトン描画」、サンプル 10「顔絵文字」などの Canvas 描画技術の基礎。

B.2.5 プロンプト 5：handPose の座標システム

```
ml5.jsのhandPoseが返す21個のキーポイントに
ついて教えてください。

- 各キーポイントの名前と指との対応
- x, y, z座標の意味
- 信頼度スコアの活用方法
```

ガイドとの関連: サンプル 05「手検出基本」で学ぶ handPose の詳細理解。

B.2.6 プロンプト 6：faceMesh のランドマーク

```
ml5.jsのfaceMeshが検出する468〜478個の
```

```
顔ランドマークについて教えてください。

特に以下の部位のインデックス番号を
知りたいです：
- 目の輪郭（左右それぞれ）
- 口の輪郭
- 鼻の位置
- 眉毛の位置
```

ガイドとの関連: サンプル 06「顔メッシュ基本」、サンプル 10「顔絵文字」での faceMesh 活用。

B.2.7 プロンプト 7：音声分類の仕組み

```
ml5.jsのsoundClassifierと
SpeechCommands18wモデルについて
教えてください。

- 認識できる18個の音声コマンド一覧
- 「background noise」の意味
- 認識精度を上げるコツ
```

ガイドとの関連: サンプル 07「音声分類基本」で学ぶ音声認識の理論的背景。

B.2.8 プロンプト 8：リアルタイム処理の基本

```
Webカメラの映像をリアルタイムで
処理するJavaScriptコードの書き方を
教えてください。

requestAnimationFrameを使った
ループ処理の基本パターンと、
なぜsetIntervalではなく
requestAnimationFrameを使うのかを
説明してください。
```

ガイドとの関連: 全リアルタイムサンプル共通のアニメーションループ理解。

B.2.9 プロンプト 9：エラーハンドリング

```
ml5.jsでよく発生するエラーと
その対処法を教えてください。

特に以下のケースについて：
- カメラアクセスが拒否された場合
- モデルの読み込みに失敗した場合
- 検出結果が空の場合
```

ガイドとの関連: 全サンプルでの堅牢なコード作成に必要な知識。

B.2.10 プロンプト 10：ファイルアップロード処理

```
HTMLのinput type="file"を使って
画像をアップロードし、
ml5.jsで分類するコードを教えてください。

FileReaderを使った画像読み込みの
基本パターンも含めてください。
```

ガイドとの関連: サンプル 09「画像アップロード分類」の実装理解。

B.3 中級プロンプト集

以下のプロンプトは、ml5.js の基本を理解し、より高度な応用に挑戦したい方向けです。本ガイドの第 2 章（中級編）の内容と連携しています。

B.3.1 プロンプト 11：ゲームループの設計

```
ml5.jsのbodyPoseを使った
シンプルなゲームを作りたいです。

以下の要素を含むゲームループの
設計パターンを教えてください：
- スコア管理
- 衝突判定
- ゲーム状態（開始/プレイ中/終了）
- requestAnimationFrameの適切な使い方
```

ガイドとの関連: サンプル 11「フルーツキャッチゲーム」、サンプル 27「ポーズマッチングゲーム」のゲーム設計。

B.3.2 プロンプト 12：描画アプリの実装

```
ml5.jsのhandPoseを使って、
手の動きで絵を描くアプリを作りたいです。

以下の機能の実装方法を教えてください：
- 人差し指の軌跡を描画する
- 描画中/非描画の切り替え
  （特定の指ジェスチャーで制御）
- 線の太さと色の変更
- キャンバスのクリア
```

ガイドとの関連: サンプル 12「手で描くお絵かき」の高度な実装。

B.3.3 プロンプト 13：フェイスフィルターの作成

```
ml5.jsのfaceMeshを使って、
InstagramやSnapchatのような
顔フィルターを作りたいです。

以下を実装するコードを教えてください：
- 目の位置に画像を重ねる
- 顔の傾きに合わせて画像を回転
- 顔のサイズに合わせて画像をスケール
```

ガイドとの関連: サンプル 13「フェイスフィルター」の AR 機能実装。

B.3.4 プロンプト 14：Neural Network 分類

```
ml5.jsのneuralNetworkを使って、
色を「暖色」「寒色」「中間色」に
分類するモデルを訓練したいです。

以下を含むコードを教えてください：
- RGB値を入力として使う方法
- 分類用のデータ構造
- 訓練パラメータの設定
- 訓練済みモデルでの予測
```

ガイドとの関連: サンプル 14「Neural Network 色分類」の実装手法。

B.3.5 プロンプト 15：回帰問題の実装

```
ml5.jsのneuralNetworkを回帰タスクに
使う方法を教えてください。

例として、部屋の広さと築年数から
家賃を予測するモデルを作りたいです。
分類と回帰の違いも説明してください。
```

ガイドとの関連: サンプル 15「住宅価格予測」での回帰モデル実装。

B.3.6 プロンプト 16：ポーズと音の連携

```
ml5.jsのbodyPoseとWeb Audio APIを
組み合わせて、体の動きで音を
コントロールするアプリを作りたいです。
```

```
4
5  以下の実装方法を教えてください：
6  - 手の位置でピッチを変える
7  - 体の開き具合で音量を変える
8  - OscillatorNodeの使い方
```

ガイドとの関連: サンプル 16「体で音楽を奏でる」のサウンド制御技術。

B.3.7 プロンプト 17：ジェスチャー認識の設計

```
1  ml5.jsのhandPoseを使って、
2  カスタムジェスチャーを認識する
3  システムを設計したいです。
4
5  以下のジェスチャーの判定ロジックを
6  教えてください：
7  - グー、チョキ、パー
8  - 親指を立てる（いいね）
9  - ピースサイン
10 - 指でスワイプ
```

ガイドとの関連: サンプル 17「ジェスチャー認識ゲーム」の判定アルゴリズム。

B.3.8 プロンプト 18：複数モデルの統合

```
1  ml5.jsで複数のモデル（bodyPose、
2  handPose、faceMesh）を同時に
3  使用する方法を教えてください。
4
5  以下の点について説明してください：
6  - 各モデルの初期化順序
7  - パフォーマンスへの影響
8  - 検出結果の統合方法
9  - フレームレートの最適化
```

ガイドとの関連: サンプル 18「マルチモデル検出」の設計パターン。

B.3.9 プロンプト 19：音声の可視化

```
1  ml5.jsのsoundClassifierの認識結果を
2  リアルタイムで可視化するアプリを
3  作りたいです。
4
5  以下を含めてください：
6  - 認識されたラベルの表示
7  - 信頼度のバーグラフ表示
```

```
- 時系列での認識履歴
- Canvasでのアニメーション
```

ガイドとの関連: サンプル 19「音声ビジュアライザー」の実装手法。

B.3.10 プロンプト 20：カスタム UI の設計

```
ml5.jsの画像分類アプリに、
使いやすいカスタムUIを追加したいです。

以下のUI要素の実装を教えてください：
- ドラッグ&ドロップでの画像アップロード
- 分類結果のカード形式表示
- ローディングインジケーター
- レスポンシブデザイン
```

ガイドとの関連: サンプル 20「カスタム UI 画像分類」の UI/UX 設計。

B.4 上級プロンプト集

以下のプロンプトは、ml5.js を実践的なプロジェクトで活用したい上級者向けです。本ガイドの第 3 章（上級編）の内容と連携しています。

B.4.1 プロンプト 21：カスタムポーズ分類器

```
ml5.jsのbodyPoseとneuralNetworkを
組み合わせて、カスタムポーズを
分類するシステムを作りたいです。

以下の点について詳しく教えてください：
- ポーズデータの収集方法
- 特徴量（キーポイント座標）の正規化
- 転移学習との違い
- 訓練データ数の目安
```

ガイドとの関連: サンプル 21「カスタムポーズ分類器」の機械学習パイプライン。

B.4.2 プロンプト 22：フィットネストラッカー

```
ml5.jsを使って、エクササイズの
回数をカウントするフィットネス
トラッカーを作りたいです。

以下の運動のカウント方法を
教えてください：
```

```
- スクワット（膝の角度変化）
- 腕立て伏せ（肘の角度変化）
- ジャンピングジャック（手足の開き）

状態マシンによる
カウントロジックも含めてください。
```

ガイドとの関連: サンプル 22「フィットネストラッカー」の運動検出アルゴリズム。

B.4.3 プロンプト 23：バーチャルキーボード

```
ml5.jsのhandPoseを使って、
空中で文字入力できるバーチャル
キーボードを実装したいです。

以下の機能を含めてください：
- キーボードレイアウトの描画
- 指先とキーの衝突判定
- タップ判定（z座標の変化）
- 入力確定のフィードバック
```

ガイドとの関連: サンプル 23「バーチャルキーボード」の空間インタラクション。

B.4.4 プロンプト 24：頭の向き推定

```
ml5.jsのfaceMeshを使って、
頭の向き（yaw, pitch, roll）を
推定する方法を教えてください。

以下を含めてください：
- 使用するランドマークポイント
- 3D回転角度の計算方法
- 視線追跡への応用
- 居眠り検知への応用
```

ガイドとの関連: サンプル 24「頭の向き推定」の 3D 姿勢推定技術。

B.4.5 プロンプト 25：複数人トラッキング

```
ml5.jsで複数人のポーズを同時に
トラッキングするシステムを
作りたいです。

以下の課題への対処法を
教えてください：
- 人物の識別（ID管理）
```

```
- 人物の入退場処理
- パフォーマンス最適化
- 交差や重なりへの対応
```

ガイドとの関連: サンプル 25「マルチパーソントラッキング」の識別システム。

B.4.6 プロンプト 26：時系列予測

```
ml5.jsのneuralNetworkを使って、
時系列データの予測を行いたいです。

株価のような連続データを予測する
ための以下の手法を教えてください：
- スライディングウィンドウの作成
- シーケンスデータの正規化
- LSTM/RNNとの違い
- 予測精度の評価方法
```

ガイドとの関連: サンプル 26「時系列予測」のシーケンスモデリング。

B.4.7 プロンプト 27：ポーズマッチング

```
ml5.jsを使って、お手本のポーズと
ユーザーのポーズの類似度を
計算するシステムを作りたいです。

以下のアルゴリズムを教えてください：
- ポーズの正規化方法
- 類似度スコアの計算
  （コサイン類似度、ユークリッド距離）
- スコアに基づくフィードバック
```

ガイドとの関連: サンプル 27「ポーズマッチングゲーム」のマッチングアルゴリズム。

B.4.8 プロンプト 28：指文字認識

```
ml5.jsのhandPoseとneuralNetworkを
使って、アメリカ手話（ASL）の
指文字を認識するシステムを
作りたいです。

以下を含めてください：
- 指文字の特徴量設計
- データ収集のUI
- リアルタイム認識の実装
- 認識精度向上のコツ
```

ガイドとの関連: サンプル 28「指文字認識」のジェスチャー分類システム。

B.4.9 プロンプト 29：AI お絵かきアシスタント

```
ml5.jsのimageClassifierを使って、
ユーザーの描いた絵を認識し、
フィードバックを返すお絵かき
アシスタントを作りたいです。

以下の機能を実装したいです：
- フリーハンド描画キャンバス
- 描画内容のリアルタイム分類
- 分類結果に基づくヒント表示
- 描画履歴の保存
```

ガイドとの関連: サンプル 29「AI お絵かきアシスタント」のインタラクティブ AI。

B.4.10 プロンプト 30：総合 AI システム

```
ml5.jsの複数モデルを統合した、
インタラクティブなAI体験アプリを
設計したいです。

以下を含む総合システムの
アーキテクチャを提案してください：
- 体、手、顔、音声の統合検出
- 各入力に応じたビジュアル反応
- パフォーマンス最適化戦略
- 状態管理パターン
- 拡張性を考慮した設計
```

ガイドとの関連: サンプル 30「総合 AI インタラクティブ体験」のシステム設計。

B.5 プロンプト活用のベストプラクティス

B.5.1 段階的な学習アプローチ

1. **理解フェーズ**: 概念や仕組みを質問して理解を深める
2. **模倣フェーズ**: サンプルコードを動かして動作を確認する
3. **改造フェーズ**: パラメータを変えて実験する
4. **創造フェーズ**: オリジナルの機能を追加する

B.5.2 トラブルシューティング用プロンプト

```
以下のml5.jsコードでエラーが発生します。
原因と解決策を教えてください。

[エラーメッセージ]
（ここにコンソールのエラーを貼り付け）

[コード]
（ここに問題のコードを貼り付け）

[環境]
- ブラウザ: Chrome 最新版
- ml5.js: v1.0
- 実行環境: ローカルサーバー
```

B.5.3 コードレビュー用プロンプト

```
以下のml5.jsコードをレビューしてください。
改善点があれば指摘してください。

特に以下の観点でお願いします：
- パフォーマンス
- コードの可読性
- エラーハンドリング
- ベストプラクティスとの比較

[コード]
（ここにレビューしてほしいコードを貼り付け）
```

拡張のヒント

これらのプロンプトをカスタマイズして、自分のプロジェクトに合わせた質問を作成してみましょう。具体的なコードや状況を含めるほど、より的確な回答が得られます。

索引