多层感知机 (MLP) 实现文档

这是一个从零开始实现的简单多层感知机 (MLP) 库，旨在提供一个清晰、模块化的深度学习基础组件。该库包含了构建、训练和评估 MLP 模型所需的核心功能，并支持分类和回归任务。

核心组件

1. `mlp.py` ()

该文件定义了多层感知机的核心 类。

- **`MLP` 类**:
  - **初始化 (`__init__`)**: 允许用户指定网络层数 (`layers`)、每层的神经元数量、激活函数 (`activations`)、权重初始化方法 (`weight_initializer`)等。权重和偏置在此阶段被初始化。

```
class MLP:
    def __init__(self, layers, activations,
weight_initializer='zeros'):
        self.layers = layers
        self.activations = [get_activation(act) for act in
activations]
        self.weights = []
        self.biases = []
        self.weight_initializer =
get_initializer(weight_initializer)
        self._initialize_weights()
```

  - **权重初始化 (`_initialize_weights`)**: 根据指定的初始化策略（如 Xavier, He）为网络的每一层创建权重矩阵和偏置向量。

```
    def _initialize_weights(self):
        for i in range(len(self.layers) - 1):
            weight =
self.weight_initializer.initialize((self.layers[i],
self.layers[i+1]))
            bias = np.zeros((1, self.layers[i+1]))
            self.weights.append(weight)
            self.biases.append(bias)
```

- **前向传播 (`forward`)**: 接收输入数据 X，逐层计算加权和 ((Z = W \cdot A_{prev} + b)) 和激活输出 ((A = activation(Z)))，直到输出层。过程中会存储每一层的输入 (`layer_inputs`) 和激活前的输出 (`layer_outputs_pre_activation`)，供反向传播使用。

```
def forward(self, X):
    self.layer_inputs = [X]
    self.layer_outputs_pre_activation = []
    A = X
    for i in range(len(self.layers) - 1):
        Z = np.dot(A, self.weights[i]) + self.biases[i]
        self.layer_outputs_pre_activation.append(Z)
        A = self.activations[i].forward(Z)
        self.layer_inputs.append(A)
    return A
```

- **反向传播 (`backward`)**: 从输出层开始，根据损失函数对输出层激活的梯度 `dL_dA`，利用链式法则逐层计算损失函数对权重 (`dW`)、偏置 (`db`) 以及前一层激活 (`dA_prev`) 的梯度。

```
def backward(self, dL_dA):
    gradients = []
    d_prev = dL_dA
    num_layers = len(self.layers) - 1
    for i in reversed(range(num_layers)):
        A_prev = self.layer_inputs[i]
        Z_curr = self.layer_outputs_pre_activation[i]
        current_activation = self.activations[i]
        dZ = current_activation.backward(Z_curr, d_prev)
        dW = np.dot(A_prev.T, dZ)
        db = np.sum(dZ, axis=0, keepdims=True)
        d_prev = np.dot(dZ, self.weights[i].T)
        gradients.insert(0, (dW, db))
    return gradients
```

- **参数更新 (`update_weights`)**: 使用选定的优化器（如 SGD, Adam）和计算得到的梯度来更新模型的权重和偏置。

```
def update_weights(self, gradients, learning_rate, optimizer):
    self.weights, self.biases = optimizer.update(self.weights,
self.biases, gradients, learning_rate)
```

- **训练 (`train`)**: 整合了前向传播、损失计算、反向传播和参数更新的完整训练循环。支持批量训练、周期 (epochs) 控制、早停、正则化等功能。
- **预测 (`predict`)**: 使用训练好的模型对新数据进行前向传播，得到预测结果。

2. `activations.py` ()

此文件定义了各种激活函数及其导数，它们都继承自基类 。

- **`Activation` 基类**: 定义了 `forward` 和 `backward` 方法的接口。
- **ReLU ()**: Rectified Linear Unit。

    - `forward(Z)`: 计算 (max(0, Z))。
    - `backward(Z_cache, dA)`: 计算梯度，如果 (Z > 0) 则为 `dA`，否则为 0。

```
class ReLU(Activation):
    def forward(self, Z):
        self.cache = Z
        return np.maximum(0, Z)
    def backward(self, Z_cache, dA):
        dZ = np.array(dA, copy=True)
        dZ[Z_cache <= 0] = 0
        return dZ
```

- **Sigmoid ()**: Sigmoid 函数。
    - `forward(Z)`: 计算 (1 / (1 + e^{-Z}))。
    - `backward(A_cache, dA)`: 计算梯度，(dA \cdot s \cdot (1 - s))，其中 (s) 是 Sigmoid 的输出。

```
class Sigmoid(Activation):
    def forward(self, Z):
        self.cache = 1 / (1 + np.exp(-Z))
        return self.cache
    def backward(self, Z_cache_is_A, dA):
        s = Z_cache_is_A
        dZ = dA * s * (1 - s)
        return dZ
```

- **Softmax ()**: Softmax 函数，常用于多分类任务的输出层，将输出转换为概率分布。
    - `forward(Z)`: 计算 (e^{Z_i} / \sum_j e^{Z_j})。
    - `backward(A_cache, dL_dA)`: 当与交叉熵损失结合时，其梯度计算通常被简化，这里假设 `dL_dA` 已经是简化后的 (dL/dZ)。
- **Linear ()**: 线性激活函数（即无激活）。
    - `forward(Z)`: 直接返回 (Z)。
    - `backward(Z_cache, dA)`: 直接返回 `dA`。
- **`get_activation(name)` ()**: 一个工厂函数，根据名称返回相应的激活函数对象。

3. `losses.py` ()

该文件包含了用于评估模型预测与真实标签之间差异的损失函数，它们都继承自基类 。

- **`Loss` 基类**: 定义了 `compute_loss` 和 `backward` 方法的接口。
- **均方误差 (Mean Squared Error, MSE) ()**: 常用于回归任务。
    - `compute_loss(y_true, y_pred)`: 计算 (\frac{1}{N} \sum (y_{true} - y_{pred})^2)。
    - `backward(y_true, y_pred)`: 计算损失对 `y_pred` 的梯度，即 (2(y_{pred} - y_{true}) / N)。

```
class MeanSquaredError(Loss):
    def compute_loss(self, y_true, y_pred):
        return np.mean((y_true - y_pred)**2)
```

```
    def backward(self, y_true, y_pred):
        return 2 * (y_pred - y_true) / y_true.shape[0]
```

- **交叉熵损失 (Cross-Entropy Loss) ()**: 常用于分类任务，特别是与 Softmax 输出层结合使用。
  - `compute_loss(y_true, y_pred)`: 对于多分类（`y_true` 是 one-hot 编码），计算 (-\frac{1}{N} \sum y_{true} \log(y_{pred}))。支持 `y_true` 为类别索引的情况。
  - `backward(y_true, y_pred)`: 计算损失对 `y_pred` 的梯度。当与 Softmax 结合时，梯度简化为 ((y_{pred} - y_{true}) / N)（假设 `y_true` 是 one-hot 编码）。

```
class CrossEntropyLoss(Loss):
    def compute_loss(self, y_true, y_pred):
        y_pred = np.clip(y_pred, 1e-12, 1 - 1e-12)
        if y_true.shape == y_pred.shape:
            loss = -np.sum(y_true * np.log(y_pred)) / y_true.shape[0]
        else:
            num_samples = y_true.shape[0]
            log_likelihood = -np.log(y_pred[range(num_samples),
y_true])
            loss = np.sum(log_likelihood) / num_samples
        return loss
    def backward(self, y_true, y_pred):
        if y_true.shape != y_pred.shape:
            num_samples = y_true.shape[0]
            num_classes = y_pred.shape[1]
            y_true_one_hot = np.zeros((num_samples, num_classes))
            y_true_one_hot[np.arange(num_samples), y_true] = 1
            y_true = y_true_one_hot
        return (y_pred - y_true) / y_true.shape[0]
```

- `get_loss_function(name)` **()**: 一个工厂函数，根据名称返回相应的损失函数对象。

4. `optimizers.py`

实现了多种优化算法，用于更新模型参数：

- **随机梯度下降 (Stochastic Gradient Descent, SGD)**: `SGD` 类
- **Adam**: `Adam` 类

5. `initializers.py`

提供了不同的权重初始化策略，以帮助模型更好地收敛：

- **零初始化**: `zeros_init`
- **随机初始化**: `random_init`
- **Xavier/Glorot 初始化**: `xavier_init`
- **He 初始化**: `he_init`

6. `regularization.py`

包含了正则化技术，用于防止过拟合：

- **L1 正则化**: `l1_regularization`, `l1_regularization_derivative`
- **L2 正则化**: `l2_regularization`, `l2_regularization_derivative`

7. `metrics.py`

提供了评估模型性能的指标：

- **混淆矩阵 (Confusion Matrix)**: `confusion_matrix`
- **准确率 (Accuracy Score)**: `accuracy_score`
- **精确率 (Precision Score)**: `precision_score`
- **召回率 (Recall Score)**: `recall_score`
- **F1 分数 (F1 Score)**: `f1_score`

8. `utils.py`

包含了一些实用工具函数：

- **独热编码 (One-Hot Encoding)**: `to_one_hot`
- **训练集/测试集划分**: `train_test_split`
- **批量数据生成**: `get_batches`

9. `api.py`

- **`MLPAPI` 类**: 提供了一个高级 API 接口，简化了 MLP 模型的训练、预测、评估、保存和加载过程。它封装了底层 MLP 类的复杂性，使得用户可以更便捷地使用模型。

10. `__init__.py`

- 作为 Python 包的初始化文件，它定义了 `MLP/src` 目录下的模块结构，并使用 `__all__` 列表导出了核心类和函数，方便用户直接从 `MLP.src` 导入。

设计原则

- **模块化**: 每个核心功能（如激活函数、损失函数、优化器）都被封装在独立的模块中，提高了代码的可读性和可维护性。
- **可扩展性**: 易于添加新的激活函数、损失函数、优化器或正则化方法。
- **灵活性**: 用户可以根据需求自定义网络结构、选择不同的组件。
- **Numpy 实现**: 核心计算基于 NumPy，避免了对大型深度学习框架的依赖，有助于理解底层机制。

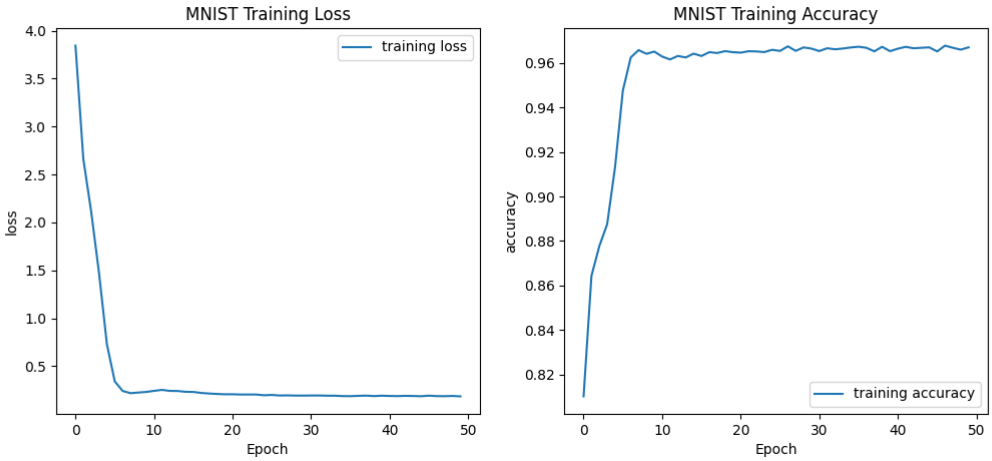
训练与评估结果

我们在两个经典的数据集上对 MLP 实现进行了测试：MNIST 手写数字分类和 California Housing 房价回归。

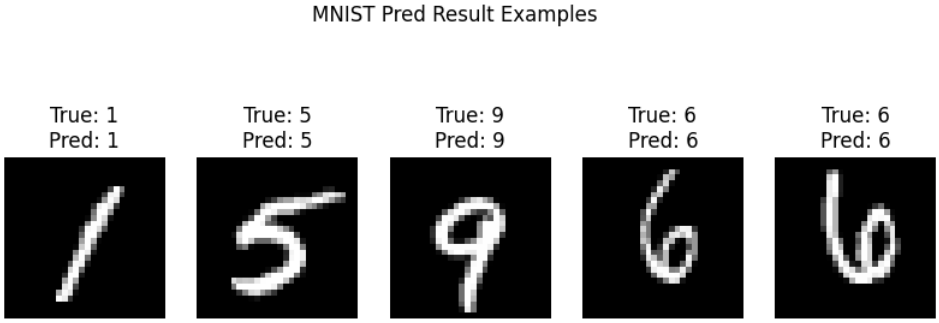
1. MNIST 手写数字分类

- **数据集**: MNIST 手写数字数据集 (60,000 训练样本, 10,000 测试样本, 10 个类别)。
- **模型配置**:
  - 输入层: 784 (28x28 像素)
  - 隐藏层: 256 个神经元 (ReLU 激活)

    - 输出层: 10 个神经元 (Softmax 激活)
    - 损失函数: 交叉熵
    - 优化器: Adam
    - 训练轮次: 50
    - 批量大小: 64
- **结果**:
    - 训练损失和准确率随训练轮次的变化曲线（图像应位于项目根目录，名为 `mnist_training_history.png`）：



    - 随机选择的 MNIST 预测示例（图像应位于项目根目录，名为 `mnist_predictions_example.png`）：



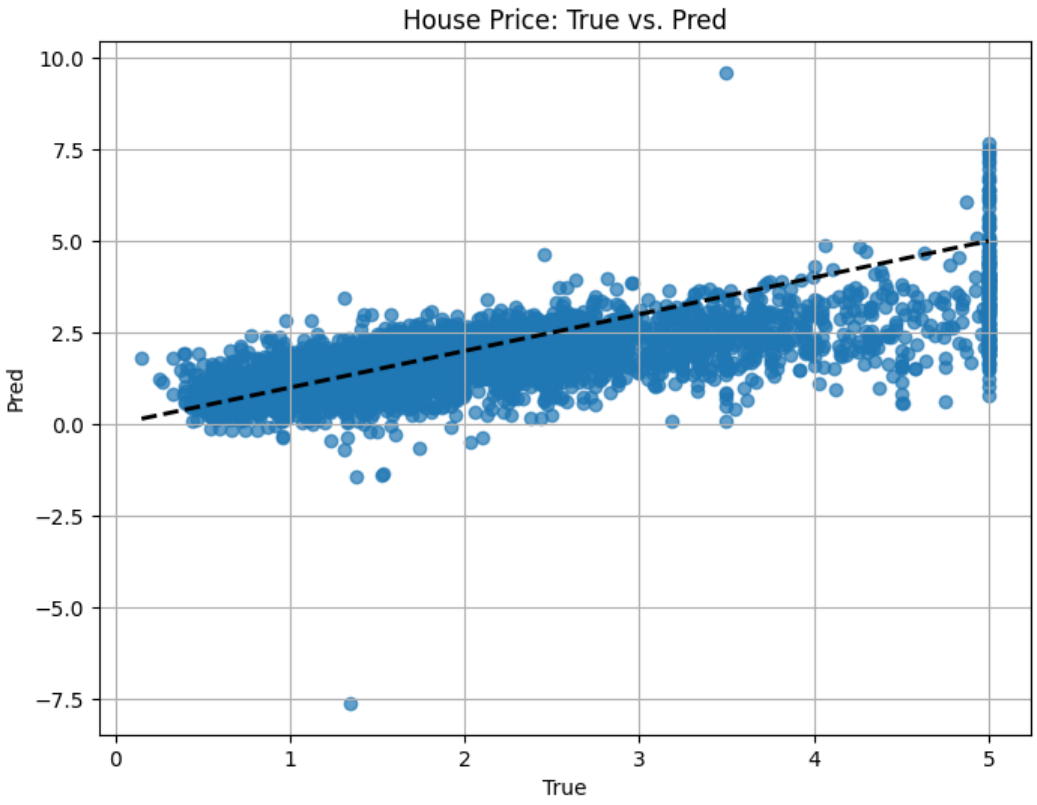
    - 模型在测试集上的准确率约为 97%。
    - 混淆矩阵展示了模型在各个数字类别上的分类性能。

2. California Housing 回归

- **数据集**: California Housing 数据集 (20,640 样本，8 个特征，预测房价中位数)。
- **模型配置**:
    - 输入层: 8 个特征
    - 隐藏层: 64 个神经元 (ReLU 激活)
    - 输出层: 1 个神经元 (无激活函数)
    - 损失函数: 均方误差 (MSE)
    - 优化器: Adam
    - 训练轮次: 100
    - 批量大小: 32

- **结果**:
  - 训练损失随训练轮次的变化曲线（图像应位于项目根目录，名为 `california_training_history.png`）：

  - 真实值与预测值的散点图（图像应位于项目根目录，名为 `california_predictions_scatter.png`），显示了模型对房价的预测能力：



- 模型在测试集上的 MSE 较低，表明回归效果良好。

模型保存与加载

- `MLPAPI` 提供了 `save_model` 和 `load_model` 方法，可以将训练好的模型参数保存到 `.pkl` 文件中，并在需要时重新加载，方便模型的部署和复用。

关键设计与实现

本项目在设计与实现多层感知机时，重点关注了以下几个方面，以确保代码的效率、可读性和易用性：

- **NumPy 向量化操作**: 为了提升计算效率，我们广泛采用了 NumPy 的向量化操作。例如，在计算加权和 (Z = W \cdot A + b) 或激活函数的输出时，直接对整个矩阵或向量进行操作，而不是使用 Python 的显式循环。这充分利用了 NumPy 底层优化的 C 实现，显著加快了前向传播和反向传播的计算速度。

- **精确的梯度计算**: 反向传播算法是神经网络训练的核心。我们严格遵循链式法则来计算损失函数对于网络中每一层权重和偏置的梯度。梯度从输出层开始，逐层向后传播，确保了梯度计算的准确性，这是优化算法能够有效更新参数、使模型收敛的基础。例如，对于某一层，其输出对输入的导数、激活函数对加权和的导数以及损失对该层输出的导数会相乘，以得到损失对该层加权和的导数。

- **模块化接口 ()**: 为了方便用户使用，我们设计了 类。这个高级 API 封装了模型训练 (`train`)、预测 (`predict`)、评估 (`evaluate`)、保存 (`save_model`) 和加载 (`load_model`) 的完整流程。用户无需深入了解 类的内部复杂实现，即可快速搭建和应用 MLP 模型。

- **重要的数据预处理**: 数据预处理是机器学习成功的关键步骤。示例代码中包含了：

  - **归一化/标准化**: 如对 MNIST 图像像素值进行归一化 (除以 255) 或对 California Housing 数据进行标准化 (减去均值，除以标准差)。这有助于加速模型收敛，防止梯度消失/爆炸，并确保不同尺度

的特征得到公平对待。
    - **独热编码 (One-Hot Encoding)**: 如将 MNIST 的数字标签 (0-9) 转换为 10 维的独热向量。这是多分类任务中，配合 Softmax 和交叉熵损失函数使用的标准做法。 这些步骤强调了根据数据特性进行适当预处理的重要性。

- **训练过程与结果可视化**: 我们集成了 `matplotlib` 库，用于训练过程和结果的可视化。这包括：
    - **损失曲线/准确率曲线**: 绘制训练集和验证集（如果使用）上的损失值和准确率随训练轮次 (epochs) 变化的曲线。这有助于监控模型的学习进度，判断是否存在过拟合或欠拟合。
    - **预测结果展示**: 对于分类任务，可以展示部分样本的真实标签和模型预测标签；对于回归任务，可以绘制预测值与真实值的散点图。 这些可视化工具为用户提供了直观的方式来理解和评估模型的性能。

如何运行示例

1. **安装依赖**: 确保已安装 `numpy`, `scikit-learn`, `matplotlib`。

    ```
    pip install numpy scikit-learn matplotlib
    ```

2. **下载数据集**:
    - MNIST 数据集：请手动下载并放置到 `MLP/dataset/MINIST` 目录下。
    - California Housing 数据集：`scikit-learn` 会自动下载。
3. **运行 `example.py`**:

    ```
    python MLP/example.py
    ```

    运行后，训练历史图和预测结果图将保存在项目根目录。

总结

这个 MLP 实现不仅提供了一个功能完备的神经网络模型，更重要的是，它通过清晰的模块划分和详细的文档，帮助用户深入理解多层感知机的工作原理和实现细节。通过在 MNIST 和 California Housing 数据集上的实践，展示了其在分类和回归任务中的有效性。