A2.1（5 points）尝试解释 Epoch、Iteration、Batch 几个概念及其不同，尝试说明 batch_size 的选择依据和影响。

答：

Epoch（轮次）：
一个 Epoch 表示完整的训练数据集（全部样本）在神经网络中前向传播和反向传播的一次迭代。
在一个 Epoch 中，模型会遍历整个训练数据集一次，计算损失并更新模型的参数。
Epoch 的数量通常是一个超参数，控制着模型训练的总轮次。增加 Epoch 数可以提高模型的性能，但可能会导致过拟合。
Iteration（迭代）：
一个 Iteration 表示一次参数更新的操作，通常是由一个小批次（Batch）的数据执行前向传播和反向传播后进行的参数更新。
在一个 Epoch 中，可以有多个 Iterations，每个 Iteration 处理一个 Batch 的数据。
Iteration 的数量通常是由数据集大小和 Batch 大小来决定。
Batch（批次）：
一个 Batch 表示一组训练样本，它是数据集的一个子集，用于一次前向传播和反向传播的计算。
Batch 的大小（batch size）是一个重要的超参数，控制了每次参数更新使用的样本数量。
Batch 大小的选择会直接影响训练过程的速度和性能。常见的 Batch 大小有 32、64、128 等。
综合来说：从迭代次数角度，一个 Epoch 中有许多个 Iteration。从数据量角度，一个 Epoch 中包含着全部样本，一个 Iteration 中包含着 Batch 决定的样本数量。
使用我自己在 A1 全连接神经网络作业中的代码来举例：
有 MiniBatch 训练函数如下：

```
def MiniBGD(nn_params, input_layer_size, hidden_layer_size, num_labels, X, y, lambda_reg, iter_num, alpha_rate, X_test,
            y_test):
    batch_size = 64
    m = X.shape[0]
    loss_history = []
    train_accuracy_history = []  # 用于存储训练集准确度的历史数据
    test_accuracy_history = []  # 用于存储测试集准确度的历史数据
    for i in range(iter_num):
        # 随机打乱数据和标签，以创建随机的小批次
        indices = list(range(m))
        random.shuffle(indices)
        totalcost = 0
        for j in range(0, m, batch_size):
            batch_indices = indices[j:j + batch_size]
            X_batch = X[batch_indices]
            y_batch = y[batch_indices]

            cost, grad = neural_network(nn_params, input_layer_size, hidden_layer_size, num_labels, X_batch, y_batch,
                                        lambda_reg)
            nn_params -= alpha_rate * grad
            totalcost += cost
        loss_history.append(totalcost / batch_size)
        train_accuracy, test_accuracy = accuracy(nn_params, input_layer_size, hidden_layer_size, num_labels, X, y,
```

其中Epoch和Iteration分别为大小两个循环，小循环中的batchsize即为batch：

```
    for i in range(iter_num):
        # 随机打乱数据和标签，以创建随机的小批次
        indices = list(range(m))
        random.shuffle(indices)
        totalcost = 0
        for j in range(0, m, batch_size):
            batch_indices = indices[j:j + batch_size]
            X_batch = X[batch_indices]
            y_batch = y[batch_indices]
```

Batch size 的选择依据和影响：
训练速度：较大的 Batch size 可以加速训练过程，因为每次参数更新处理更多的样本，减少了参数更新的频率。但过大的 Batch size 可能会导致内存不足或计算资源不足的问题。
泛化能力：较小的 Batch size 可以帮助模型更好地泛化，因为它在每个 Iteration 中接触到不同的数据，有助于模型更好地适应数据的多样性。但较小的 Batch size 可能会导致训练过程中的噪声，使得收敛不稳定。
超参数调整：Batch size 通常需要与学习率一起进行调整，因为它们之间有关联。较大的 Batch size 可能需要较大的学习率，而较小的 Batch size 可能需要较小的学习率，以确保参数更新的稳定性。
计算资源：Batch size 的选择还受限于可用的计算资源。在具有有限内存和计算能力的硬件上，需要选择合适的 Batch size。

A2.2 (5 points) 以一个简单的 1-1-1 结构的两层神经网络为例，分别采用均方误差损失函数和交叉熵损失函数，说明这两种函数关于参数的非凸性（可作图示意和说明）。

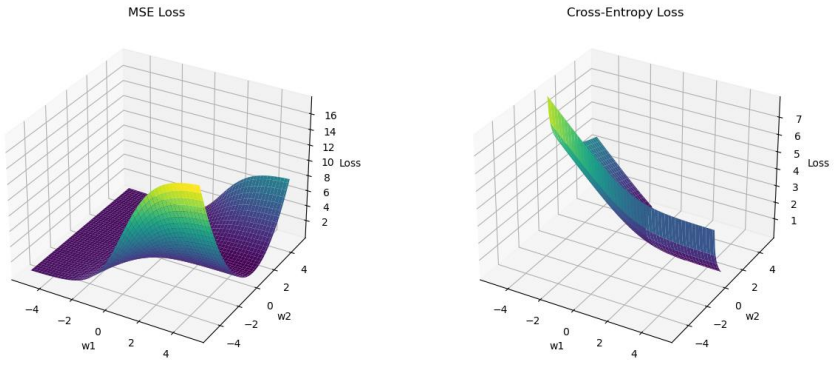
答：

写 python 代码构建神经网络，并绘图如下：

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

# 定义损失函数
def mse_loss(w1, w2):
    # 这里假设输入为 1，输出为 1
    input_data = 1
    actual_output = 1
    hidden_output = 1 / (1 + np.exp(-(w1 * input_data)))  # Sigmoid 激活函数
```

```
        predicted_output = w2 * hidden_output
        return 0.5 * (predicted_output - actual_output) ** 2

    def cross_entropy_loss(w1, w2):
        input_data = 1
        actual_output = 1
        hidden_output = 1 / (1 + np.exp(-(w1 * input_data)))  # Sigmoid 激活函数
        predicted_output = w2 * hidden_output
        return - (actual_output * np.log(predicted_output) + (1 - actual_output) * np.log(1 - predicted_output))

    # 创建参数空间
    w1_range = np.linspace(-5, 5, 100)
    w2_range = np.linspace(-5, 5, 100)
    W1, W2 = np.meshgrid(w1_range, w2_range)

    # 计算损失函数值
    MSE_loss = np.zeros_like(W1)
    CE_loss = np.zeros_like(W2)

    for i in range(len(w1_range)):
        for j in range(len(w2_range)):
            MSE_loss[i][j] = mse_loss(W1[i][j], W2[i][j])
            CE_loss[i][j] = cross_entropy_loss(W1[i][j], W2[i][j])

    # 绘制三维图像
    fig = plt.figure(figsize=(15, 6))

    # 均方误差损失函数图像
    ax1 = fig.add_subplot(121, projection='3d')
    ax1.plot_surface(W1, W2, MSE_loss, cmap='viridis')
    ax1.set_title('MSE Loss')
    ax1.set_xlabel('w1')
    ax1.set_ylabel('w2')
    ax1.set_zlabel('Loss')

    # 交叉熵损失函数图像
    ax2 = fig.add_subplot(122, projection='3d')
    ax2.plot_surface(W1, W2, CE_loss, cmap='viridis')
    ax2.set_title('Cross-Entropy Loss')
    ax2.set_xlabel('w1')
    ax2.set_ylabel('w2')
    ax2.set_zlabel('Loss')
```

```
54.  plt.show()
```



由凸函数定义可知，凸函数的一个特征是，从任意两点连成的线段在函数图像上方。由图像可发现，这两种函数关于参数具有非凸性。

A2.3 (5 points) 尝试推导：在回归问题中，假设输出中包含高斯噪音，则最小化均方误差等价于极大似然。

有线性回归模型

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p + \varepsilon$$

其中 $\beta_0 \cdots \beta_p$ 为待估计的系数

ε 为高斯噪音

$$\varepsilon \sim N(0, \delta^2)$$

最小化均方误差

$$MSE(\beta_0, \beta_1, \cdots \beta_p) = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

极大似然

由 ε 符合高斯分布，得

$$f(y_i \mid \beta_0, \beta_1, \cdots \beta_p, \delta) = \frac{1}{\sqrt{2\pi}\delta} \exp\left\{-\frac{[y_i - (\beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p)]^2}{2\delta^2}\right\}$$

最大化似然估计等于最小化似然函数负对数

$$-\log(L) = \frac{n}{2}\log(2\pi\delta^2) + \frac{1}{2\delta^2}\sum_{i=1}^{n}[Y_i - (\beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p)]^2$$

故二者等价

A2.4 (5 points) 尝试推导：在分类问题中，最小化交叉熵损失等价于极大化似然

设 y 为真实标签分布向量，如 $[0, 1, 0]$

$\hat{y}$ 为预测的概率分布，如 $[0.1, 0.7, 0.2]$

则交叉熵损失为 $H(y, \hat{y}) = -\sum_i y_i \log(\hat{y}_i)$

似然函数为 $L(\theta) = \prod_i \hat{y}_i^{y_i}$

取对数 $\log(L(\theta)) = \sum_i y_i \log(\hat{y}_i)$

因交叉熵要往小取，

似然函数要往大取，

故二者等价

A2.5 (5 points) 分析为什么 L1 正则化倾向于得到稀疏解、为什么 L2 正则化倾向于得到平滑的解。

答：

L1 正则化：
L1 正则化对模型的损失函数添加了参数的绝对值之和作为惩罚项。即：

损失 = 原始损失 + $\lambda \sum_i |w_i|$

其中，是 w 模型参数，是 lambda 正则化系数。
为什么 L1 倾向于得到稀疏解？
惩罚项鼓励模型的权重尽量小，而对于不重要的特征，L1 正则化有可能直接将其对应的权重压缩到 0。因为 L1 惩罚是不可导的（在 0 处），这导致了许多权重直接变为 0，从而得到一个稀疏解。
L2 正则化
L2 正则化对模型的损失函数添加了参数的平方和作为惩罚项。即：

损失=原始损失 + $\lambda \sum_i w_i^2$

为什么 L2 倾向于得到平滑的解？
L2 正则化鼓励模型的权重尽量小，但与 L1 正则化不同，L2 正则化不会将权重压缩到 0（除非 λ 非常大）。相反，它会将所有权重均匀地压缩，这导致了权重值都接近但不等于 0，从而得到了一个平滑的解。
为了形象理解，可以想象一个二维空间中的等高线图，L1 正则化的等高线是由菱形组成，因此优化路径可能更容易与坐标轴相交，导致某个参数变为 0；而 L2 正则化的等高线是圆形，优化路径会更趋于平滑，不容易与坐标轴相交，因此参数不容易变为 0。

L1 正则化能产生稀疏解，有助于特征选择。
L2 正则化产生非稀疏解，但可以防止过拟合，使得权重分布更平滑。

A2.6 (5 points) 分析 Batch normalization 对参数优化起到什么作用、如何起到这种作用。

答：

缓解内部协变量偏移（Internal Covariate Shift）：
在深度神经网络中，每一层的输入分布随着上一层的参数更新而变化，这会导致每一层都需要不断地适应其输入的分布变化，从而减缓训练速度。

BN 作用：通过对每个 mini-batch 进行归一化，BN 强制每一层的输入都有相似的分布，这减少了内部协变量偏移的影响。
正则化效应:
防止过拟合
BN 作用：尽管 BN 的主要目的不是正则化，但在实践中，BN 为模型引入了轻微的噪声，这有助于防止过拟合，因此，使用 BN 有时可以减少或替代其他正则化技巧如 Dropout。
梯度流:
问题：在深层网络中，梯度消失或梯度爆炸是常见问题，导致网络难以训练。
BN 作用：通过确保每一层的激活具有相似的尺度，BN 有助于保持稳定的梯度流，这样可以减轻梯度消失或梯度爆炸的问题。
允许更高的学习率:
较高的学习率可能导致训练不稳定。
BN 作用：由于 BN 提供了稳定的输入分布，并且有助于缓解梯度问题，因此它允许使用更高的学习率，从而加速训练。
减少对权重初始化的敏感性:
深度神经网络的训练对权重初始化非常敏感。
BN 作用：由于 BN 调整了各层的激活分布，这使得网络对于初始权重的选择不那么敏感，因此可以减少对特定权重初始化策略的依赖。
Batch Normalization 的核心思想是，在每一层之后加入一个归一化步骤，使得激活值均值为 0，方差为 1。然后引入两个可学习的参数（scale 和 shift），以便网络可以学习到最佳的数据分布。
Batch Normalization 通过调整和规范化每一层的输入，帮助神经网络更稳定、更快速地训练，并有助于提高模型的泛化能力。