基于 CNN 神经网络的手写字符识别实验报告

22050605 杨新莹

1. 实验目标和动机

实验的目标是使用 MNIST 数据集进行手写字符识别。MNIST 数据集包含许多手写数字的图像，每个图像都是 28x28 像素的灰度图像，表示数字 0 到 9。

实验动机是探索使用卷积神经网络（CNN）进行图像识别任务，并比较不同网络结构和参数配置对性能的影响。

输入数据： 28x28 像素的灰度图像
网络输出数据： 预测的数字类别

2. CNN 算法的基本原理

CNN 是一种专门用于处理图像数据的深度学习模型，其核心原理包括卷积层、池化层和全连接层。

卷积神经网络利用**卷积层**来提取图像特征。卷积操作通过滑动卷积核在输入图像上提取局部特征，这有助于捕获图像中的边缘、纹理等信息，并保留空间关系。

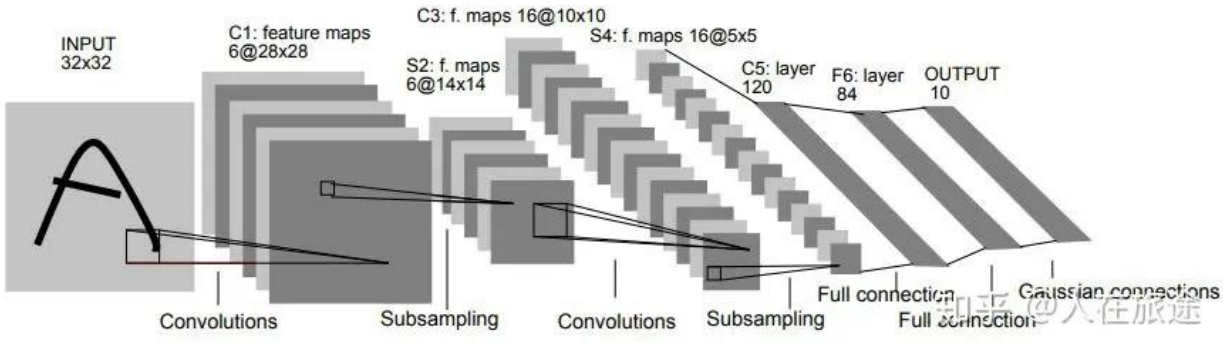
池化层用于降维和减少特征图的大小，同时保留主要特征。最常见的池化操作是最大池化，它从每个局部区域中选择最大值作为输出，减小了特征图的尺寸并提高了计算效率。

全连接层通常位于网络的顶部，将卷积层和池化层提取的特征映射到输出类别。全连接层通过学习权重参数，将特征映射到每个类别的概率，最终实现图像分类。

激活层采用激活函数，把卷积层输出结果做非线性映射。在卷积层之后使用，以增加网络的表达能力。常见的激活函数包括 ReLU（Rectified Linear Unit）、Sigmoid 和 Tanh 等。ReLU 是最常用的激活函数，它能够有效缓解梯度消失问题，并加速网络的收敛速度。

Softmax 层通常作为网络的最后一层，用于将全连接层的输出转换成各个类别的概率分布。通过对这些概率进行比较，模型可以确定输入图像最可能属于哪个类别。

3. LeNet5 网络基本架构



本实验使用 CNN 的经典模型 LeNet5 进行手写数字的识别。

LeNet-5 的基本结构包括 7 层网络结构（不含输入层），其中包括 2 个卷积层、2 个降采样层（池化层）、2 个全连接层和输出层。

1）输入层（Input layer）

输入层接收大小为 28×28 的手写数字图像，其中包括灰度值（0-255）。在本实验中对数据集进行预处理，数据进行了标准化，以加快训练速度和提高模型的准确性。[28,28,1]

2）卷积层 C1（Convolutional layer C1）

卷积层 C1 包括 6 个卷积核，每个卷积核的大小为 5×5，步长为 1，填充为 0。因此，每个卷积核会产生一个大小为 28×28 的特征图（输出通道数为 6）。[28,28,6]

3）采样层 S2（Subsampling layer S2）

采样层 S2 采用最大池化（max-pooling）操作，每个窗口的大小为 2×2 ，步长为 2。因此，每个池化操作会从 4 个相邻的特征图中选择最大值，产生一个大小为 14×14 的特征图（输出通道数为 6）。[14,14,6]

4）卷积层 C3（Convolutional layer C3）

卷积层 C3 包括 16 个卷积核，每个卷积核的大小为 5×5 ，步长为 1，填充为 0。因此，每个卷积核会产生一个大小为 10×10 的特征图（输出通道数为 16）。

[10,10,16]

5）采样层 S4（Subsampling layer S4）

采样层 S4 采用最大池化操作，每个窗口的大小为 2×2，步长为 2。因此，每个池化操作会从 4 个相邻的特征图中选择最大值，产生一个大小为 5×5 的特征图（输出通道数为 16）。[5,5,16]

6）全连接层 C5（Fully connected layer C5）

C5 将每个大小为 5×5 的特征图拉成一个长度为 400 的向量，并通过一个带有 120 个神经元的全连接层进行连接。120 是由 LeNet-5 的设计者根据实验得到的最佳值。[400,]->[120,]

7）全连接层 F6（Fully connected layer F6）

全连接层 F6 将 120 个神经元连接到 84 个神经元。[84,]

8）输出层（Output layer）

输出层由 10 个神经元组成，每个神经元对应 0-9 中的一个数字，并输出最终的分类结果。在训练过程中，使用交叉熵损失函数计算输出层的误差，并通过反向传播算法更新卷积核和全连接层的权重参数。[10,]

4. 核心架构的具体实现

```
# 构建 LeNet-5 模型
model = models.Sequential([
    layers.Conv2D(6, kernel_size=(5, 5), strides=(1, 1), activation='relu', input_shape=(28, 28, 1), padding='same'),
    layers.AveragePooling2D(pool_size=(2, 2), strides=(2, 2), padding='valid'),
    layers.Conv2D(16, kernel_size=(5, 5), strides=(1, 1), activation='relu', padding='valid'),
    layers.AveragePooling2D(pool_size=(2, 2), strides=(2, 2), padding='valid'),
    layers.Flatten(),
    layers.Dense(120, activation='relu'),
    layers.Dense(84, activation='relu'),
    layers.Dense(10, activation='softmax')
])

# 编译模型
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# 训练模型
```

```
history = model.fit(train_images, train_labels, epochs=10, batch_size=16, validation_data=(test_images, test_labels))
```

5. 网络训练和推理过程及说明

网络训练过程：

• 准备数据: 首先需要准备训练数据集和验证数据集。训练数据集通常用来训练模型的参数，验证数据集用来评估模型在训练过程中的性能。

• 构建模型: 定义 LeNet5 模型的架构，包括输入层、隐藏层、输出层以及它们之间的连接关系。这里通过深度学习框架 TensorFlow 来实现。

• 定义损失函数和优化器: 选择交叉熵作为损失函数和 Adam 作为优化器。

• 训练模型: 将模型与训练数据集进行训练，通过反向传播算法不断优化模型参数，使得模型能够更好地拟合训练数据。

• 评估模型: 使用验证数据集评估训练得到的模型的性能，观察模型在验证集上的表现，并根据评估结果进行调参和优化。

网络推理过程

• 准备输入数据: 对于需要进行推理的新数据，首先需要进行预处理，使其能够输入到训练好的模型中进行推理。

• 加载模型: 从保存的模型参数中加载训练好的模型。

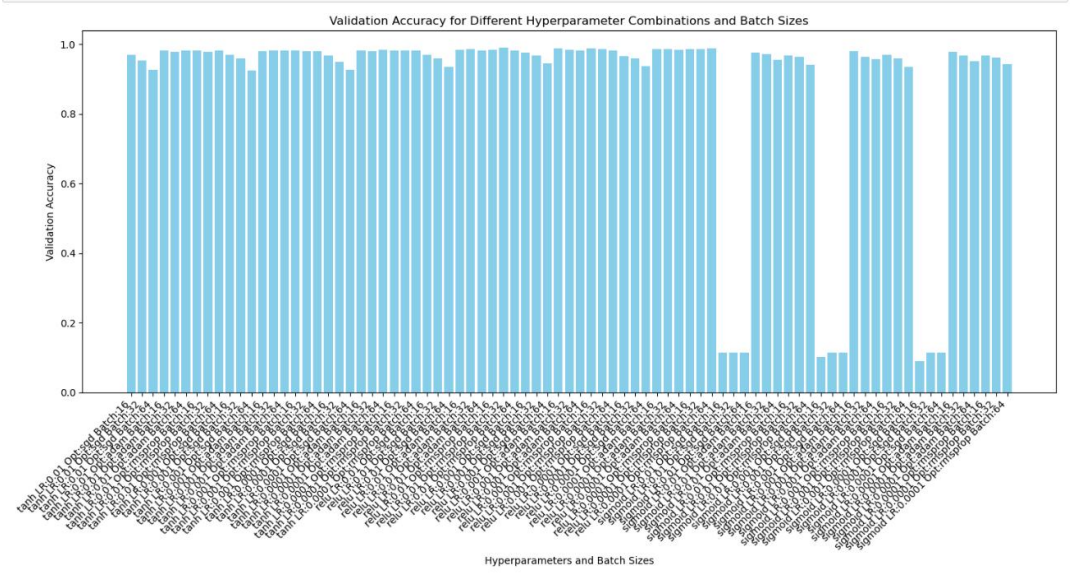
• 模型推理: 将输入数据通过加载的模型进行前向传播，得到模型的输出结果。

• 输出结果

6. 实验结果对比和分析

6.1 不同神经网络参数对性能的影响

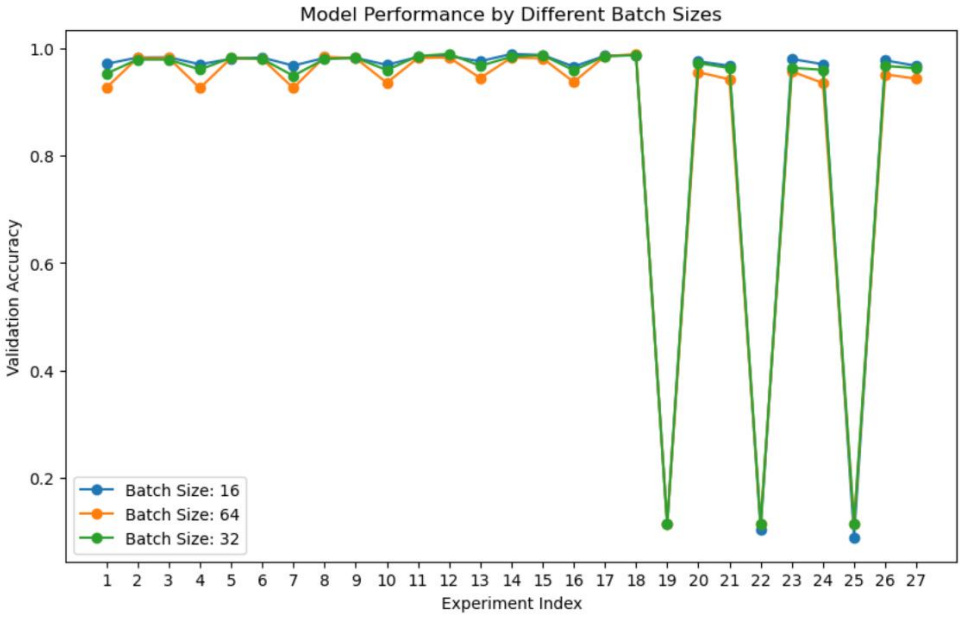
通过对网络模型的不同参数的调整，发现以下几个参数都会对模型的性能产生影响：学习率、优化算法、激活函数、批处理大小。下面是不同的模型参数对模型准确率的影响



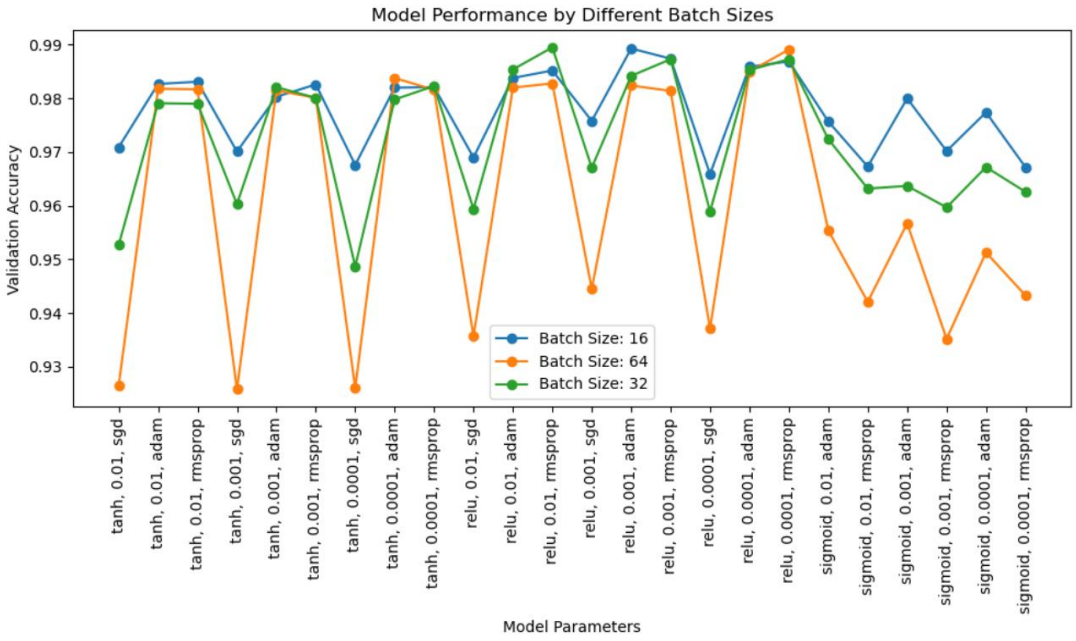
上图虽然数据过于多，但是很明显的一点是，当使用随机梯度下降（sgd）最为优化器，sigmoid 作为激活函数的时候准确率尤其低，可以发现模型的参数选择非常重要，下面是对不同模型参数的的实验结果比对和研究。

6.1.1 批处理大小

在本次实验中，我们探究了不同批量大小对 LeNet5 模型性能的影响。我们使用了批量大小为 16、32 和 64，并记录了每个批量大小下模型在测试集上的准确率。



由于有几个点的准确率过于低，使整个折现图不是很直观，于是将上述几个准确率过于低的点去掉，可以得到以下可视化折线图



我们可以明显地观察到，在不同的批量大小下，模型的准确率存在差异。具体来说，批量大小为 64 时，我们观察到最低的准确率。

一般而言，较大的批量大小可能会降低模型的收敛速度，而较小的批量大小可能导致模型更快地收敛但容易受到数据噪声的影响。这种差异可能是由于不同批量大小下梯度估计的差异所导致。

考虑到模型的性能在不同批量大小下存在明显变化，批量大小的选择在训练

深度学习模型时至关重要。然而，并不存在一种适用于所有情况的通用最佳批量大小，因此在选择批量大小时需要综合考虑模型架构、数据集特性和优化器的选择。

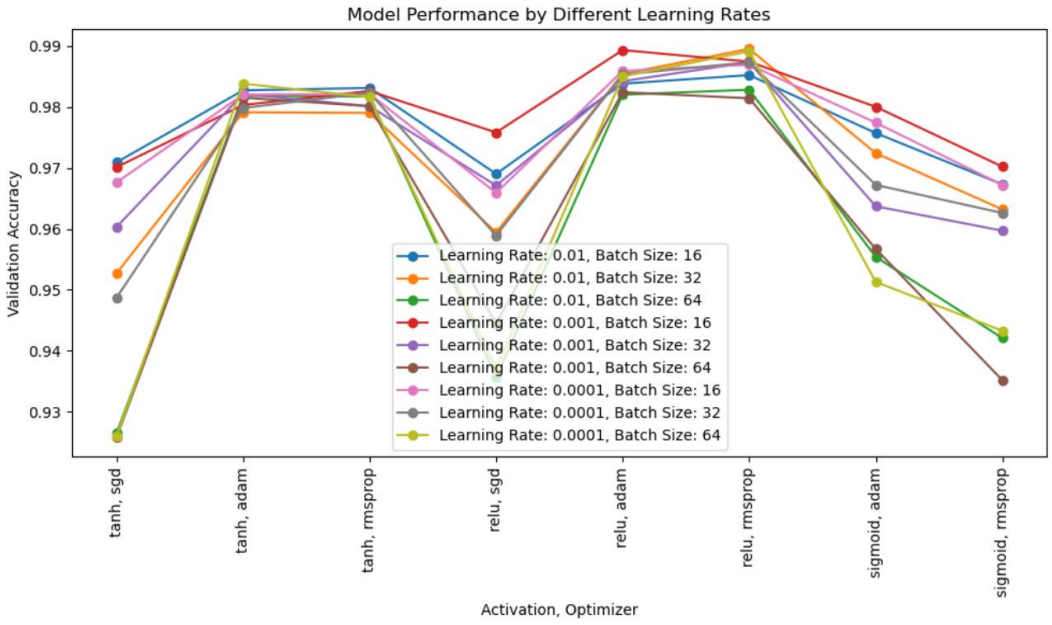
综上所述，批量大小作为一个重要的超参数，对于模型的性能具有显著影响。在实践中，我们建议根据具体情况进行实验和验证，以确定最适合的批量大小，从而有效地训练深度学习模型。

6.1.2 学习率

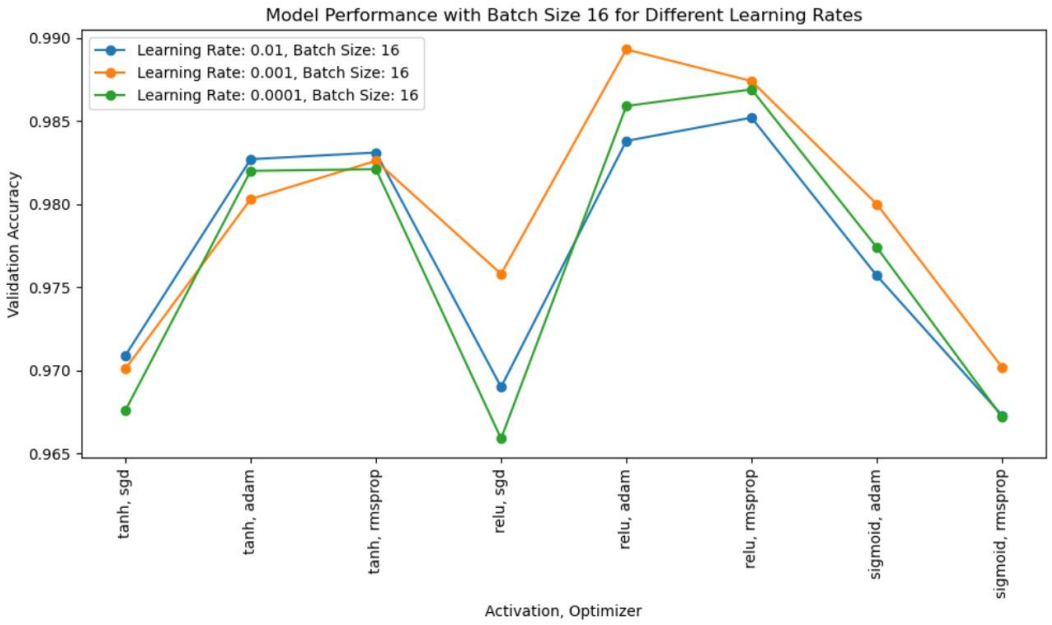
学习率（learning rate）是训练神经网络时一个非常重要的超参数，它决定了模型在每次参数更新时的步长大小。学习率的选择对模型的性能有着直接的影响，过大或过小的学习率都可能导致训练不稳定或性能下降。

如果学习率设置过大，可能导致参数更新过大，从而使得模型在参数空间中跳动幅度过大，无法收敛到最优解，甚至可能导致训练过程中出现震荡或不稳定的情况。

相反，如果学习率设置过小，模型参数更新的步长会变得很小，导致训练过程收敛速度缓慢，需要更多的迭代次数才能达到收敛，从而增加训练时间。此外，学习率过小还容易使得模型陷入局部最优解而难以获得全局最优解。



有 6.1.1 可以看出来当 batch_size 为 16 时模型性能比较好，下面只可视化 batch_size 为 16 时的模型性能



可以看出过大或过小的学习率都可能导致训练不稳定或性能下降。在模型训练时应该选择恰当的学习率。

6.1.1 优化算法

优化算法在训练神经网络时起着至关重要的作用，它决定了模型参数的更新方式和速度。不同的优化算法对模型性能有着直接的影响。

SGD 是最常见的优化算法之一，然而在本实验中，在对 MNIST 数据集的分类中 SGD 表现则没有那么良好。

相比其他两个 Adam 和 RMSprop 这两个自适应学习率算法，SGD 的表现可以说是非常差。

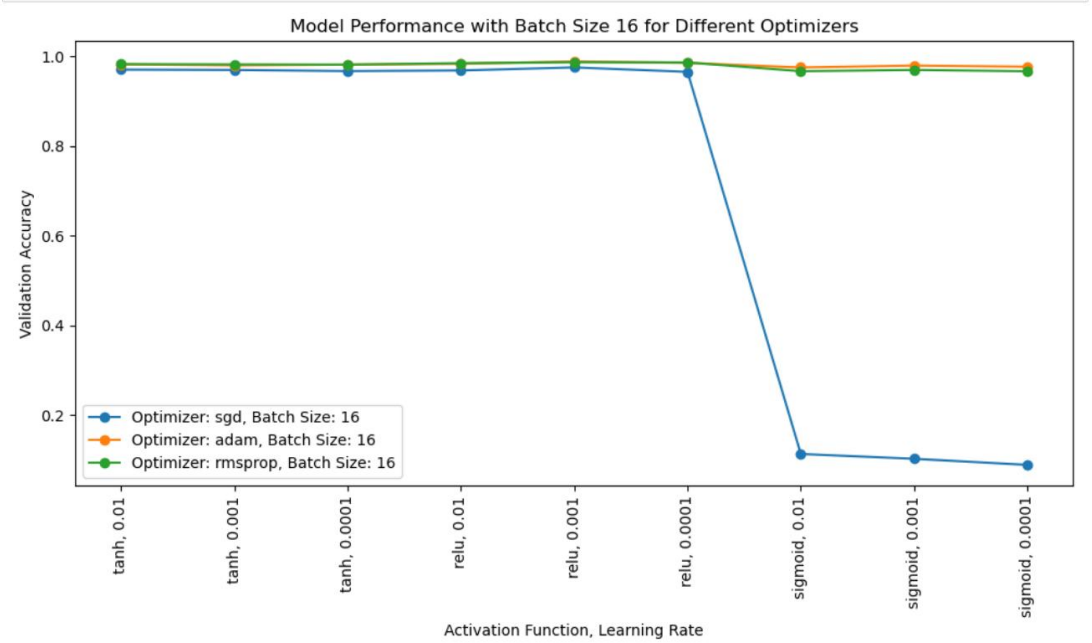
对于 MNIST 这样的任务，Adam 和 RMSprop 这类自适应学习率算法通常能够快速收敛并取得较好的性能。因为这些算法能够自动调整学习率，适应不同特征的分布和参数的尺度，从而更快地收敛到最优解。

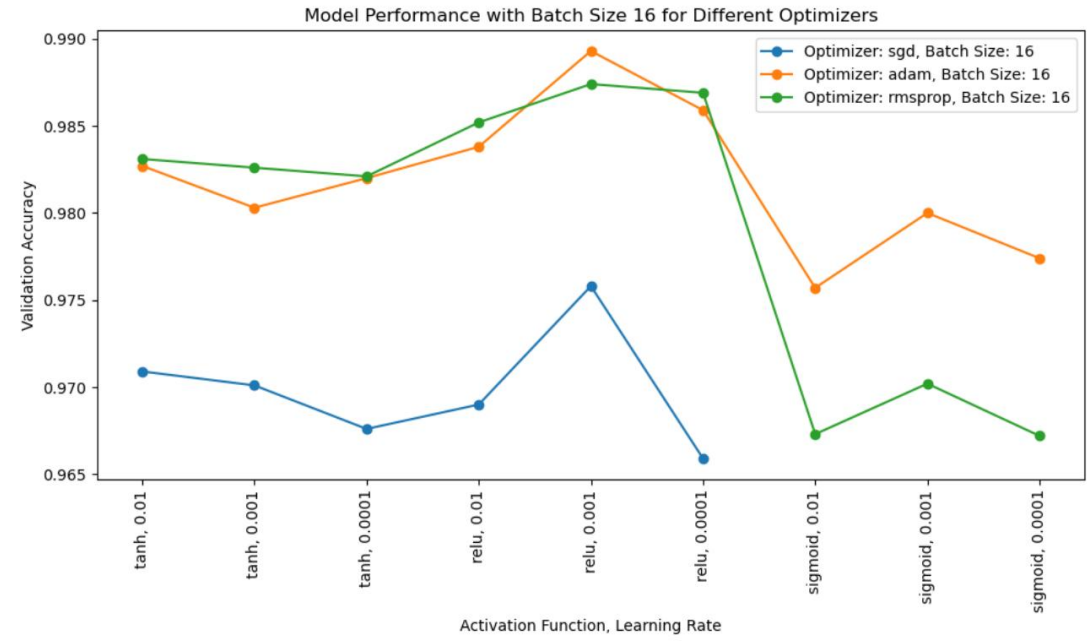
综上所述，对于 MNIST 数据集的分类任务，自适应学习率算法（如 Adam 和 RMSprop）通常能够在收敛速度和性能上取得良好的平衡。然而，对于特定的模型架构和超参数设置，不同的优化算法可能会有着不同的表现。

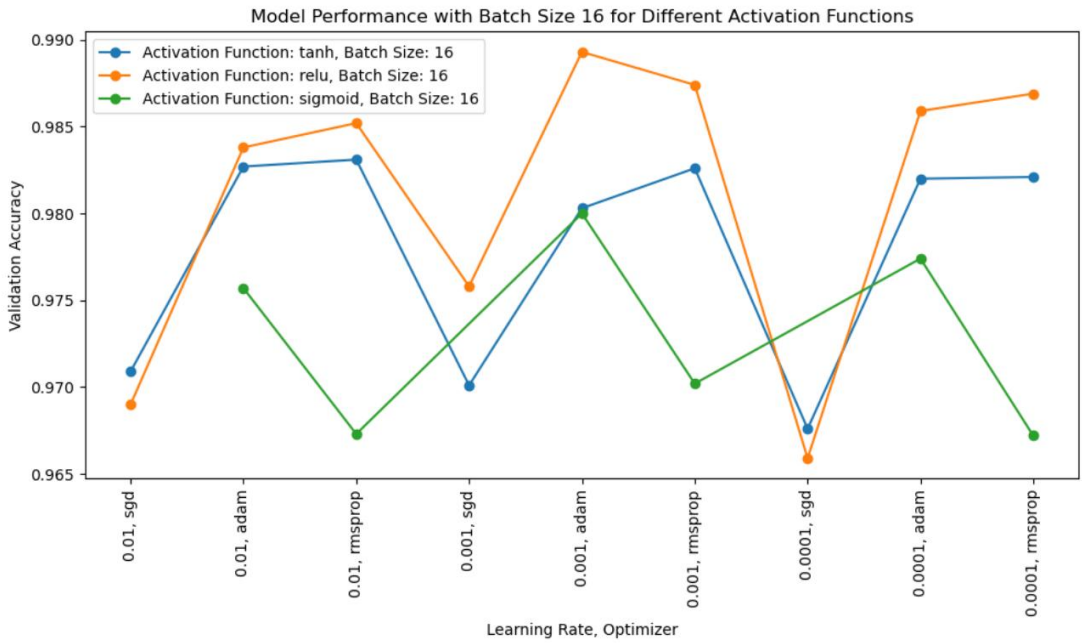
6.1.3 激活函数

不同的激活函数（如 ReLU、Sigmoid、Tanh 等）在网络训练中会对梯度传播和稀疏表示等方面产生不同的影响。实验发现就对 MNIST 数据集的分类 ReLU 表

现更为好。


Model Performance with Batch Size 16 for Different Optimizers
Validation Accuracy
1.0
0.8
0.6
0.4
0.2
Optimizer: sgd, Batch Size: 16
Optimizer: adam, Batch Size: 16
Optimizer: rmsprop, Batch Size: 16
tanh, 0.01
tanh, 0.001
tanh, 0.0001
relu, 0.01
relu, 0.001
relu, 0.0001
sigmoid, 0.01
sigmoid, 0.001
sigmoid, 0.0001
Activation Function, Learning Rate


Model Performance with Batch Size 16 for Different Optimizers
Validation Accuracy
0.990
0.985
0.980
0.975
0.970
0.965
Optimizer: sgd, Batch Size: 16
Optimizer: adam, Batch Size: 16
Optimizer: rmsprop, Batch Size: 16
tanh, 0.01
tanh, 0.001
tanh, 0.0001
relu, 0.01
relu, 0.001
relu, 0.0001
sigmoid, 0.01
sigmoid, 0.001
sigmoid, 0.0001
Activation Function, Learning Rate



由上述两个图可以看出 SGD（随机梯度下降）与 Sigmoid 激活函数结合导致准确率极为低。尝试分析了一下原因：

1）梯度消失问题： Sigmoid 函数具有饱和性，其导数在 0 到 1 之间。在网络的反向传播过程中，当梯度反向传播到较深层时，这种饱和性可能导致梯度消失，使得深层网络参数无法有效地更新，难以学习到复杂的特征表示。

2）**非线性表达受限：** Sigmoid 函数并不是非常适合作为激活函数，因为它只覆盖了一小部分输入范围，导致网络的表达能力受到限制。相比之下，ReLU 或其变体可以更好地处理非线性关系，并且不容易出现梯度消失的问题。

3）**输出分布偏移：** Sigmoid 输出值的范围在 0 到 1 之间，但是在实际应用中，可能需要更广泛的输出分布。特别是对于 MNIST 数据集中的分类问题，期望的输出是 one-hot 编码，即输出节点应该尽可能地接近 0 或 1。Sigmoid 输出的值域不是最理想的，可能导致网络训练困难。

4）**SGD 优化器的限制：** SGD 是一种简单的优化器，可能无法快速收敛到全局最优解。在复杂的模型结构中，SGD 可能会受到局部最小值或者鞍点的影响。

6.2 实验结果的分析

通过上述调试参数，最终使用如下参数进行模型的训练和测试,得到实验结果如下：

activations = ['relu']

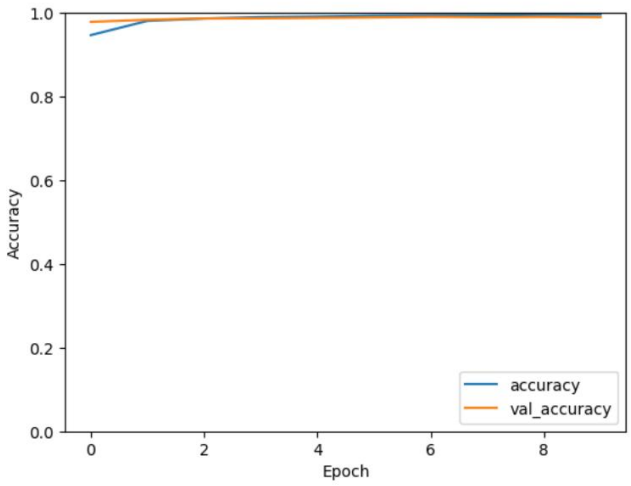
learning_rates = [0.001]

optimizers = ['adam']

batch_sizes = [16]

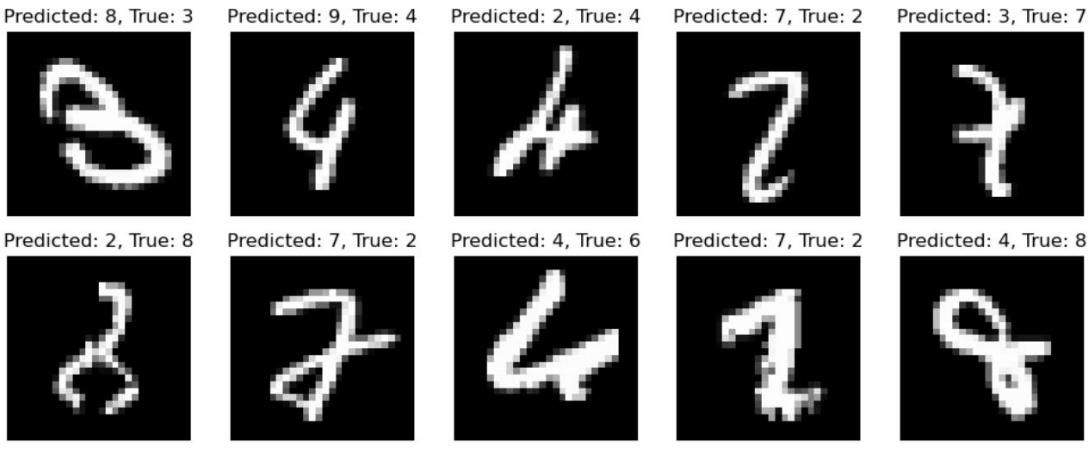
epochs = 10

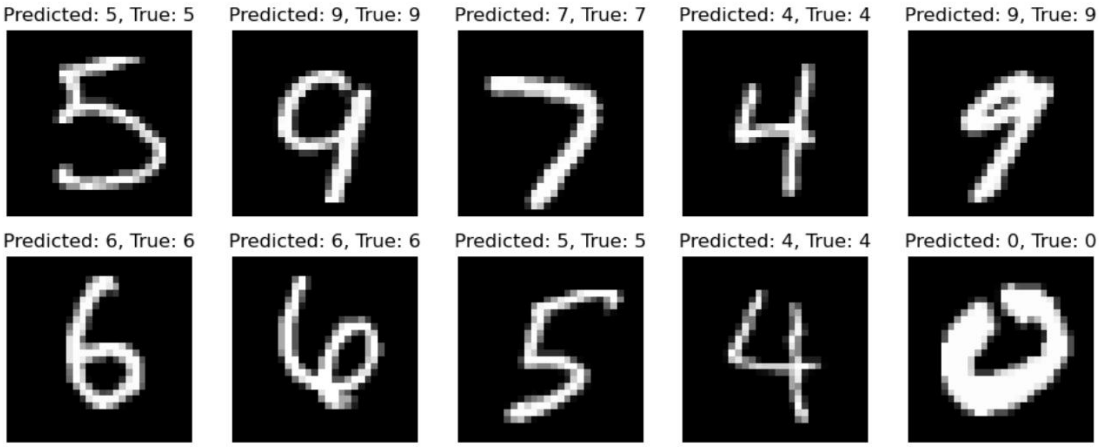
```
Epoch 1/10
3750/3750 [==============================] - 60s 15ms/step - loss: 0.1737 - accuracy: 0.9468 - val_loss: 0.0667 - val_accuracy: 0.9785
Epoch 2/10
3750/3750 [==============================] - 56s 15ms/step - loss: 0.0621 - accuracy: 0.9809 - val_loss: 0.0546 - val_accuracy: 0.9835
Epoch 3/10
3750/3750 [==============================] - 56s 15ms/step - loss: 0.0436 - accuracy: 0.9865 - val_loss: 0.0422 - val_accuracy: 0.9866
Epoch 4/10
3750/3750 [==============================] - 56s 15ms/step - loss: 0.0338 - accuracy: 0.9898 - val_loss: 0.0421 - val_accuracy: 0.9869
Epoch 5/10
3750/3750 [==============================] - 56s 15ms/step - loss: 0.0284 - accuracy: 0.9913 - val_loss: 0.0426 - val_accuracy: 0.9879
Epoch 6/10
3750/3750 [==============================] - 54s 14ms/step - loss: 0.0227 - accuracy: 0.9931 - val_loss: 0.0340 - val_accuracy: 0.9890
Epoch 7/10
3750/3750 [==============================] - 55s 15ms/step - loss: 0.0183 - accuracy: 0.9941 - val_loss: 0.0329 - val_accuracy: 0.9905
Epoch 8/10
3750/3750 [==============================] - 54s 14ms/step - loss: 0.0174 - accuracy: 0.9944 - val_loss: 0.0371 - val_accuracy: 0.9899
Epoch 9/10
3750/3750 [==============================] - 56s 15ms/step - loss: 0.0141 - accuracy: 0.9956 - val_loss: 0.0388 - val_accuracy: 0.9904
Epoch 10/10
3750/3750 [==============================] - 56s 15ms/step - loss: 0.0122 - accuracy: 0.9961 - val_loss: 0.0370 - val_accuracy: 0.9897
```




```
313/313 [==============================] - 4s 11ms/step - loss: 0.0370 - accuracy: 0.9897
Test accuracy: 0.9897000193595886
```

可视化：





可以看出模型预测出错的图片，人眼也很难识别，可见模型的训练还是比较成功的。

7. 各种神经网络方法的优缺点

7.1 多层感知机（MLP）

优点： ①简单易用，适用于一些简单的分类和回归任务。

②可以进行非线性映射，对一些非线性问题的建模能力较强。

缺点： ①处理图像、语音和自然语言处理等复杂任务的能力有限。

②受限于全连接结构，参数较多，容易过拟合。

7.2 卷积神经网络（CNN）

优点： ①在图像处理领域表现出色，能够自动提取局部特征。

②参数共享和池化操作减少了模型的参数数量，降低了过拟合风险。

缺点： ①对位置平移和尺度变化敏感，需要大量数据进行训练以获得良好的泛化性能。

②不擅长处理序列数据，如文本和时间序列数据。

7.3 循环神经网络（RNN）

优点： ①适用于处理序列数据，能够捕捉时间上的依赖关系。

②可以处理不定长的输入序列，适用于自然语言处理等任务。

缺点： ①难以捕捉长距离的依赖关系，存在梯度消失和梯度爆炸问题。

②计算效率较低，难以并行化处理。

7.4 长短期记忆网络（LSTM）和门控循环单元（GRU）

优点：①解决了传统 RNN 难以捕捉长距离依赖关系的问题。

②具有记忆单元和门控机制，能够更好地处理长序列任务。

缺点：①参数较多，需要更多的数据来训练。

②计算成本较高，对硬件要求较高。

8.总结和讨论

在 MNIST 手写字符识别任务中，CNN（例如 LeNet-5）模型表现出了较好的性能，能够有效地提取图像特征，获得较高的准确率。

参数调优对模型性能影响显著，如学习率、批处理大小、优化算法和激活函数等。具体调整这些参数能够显著地改变模型的收敛速度和准确率。

不同的优化算法和激活函数对模型性能有重要影响。自适应学习率算法（如 Adam、RMSprop）和 ReLU 激活函数通常能在训练效率和性能上取得较好的平衡。

手写字符识别问题在现代深度学习中仍然是一个重要的研究领域，不同的模型和方法在实际应用中各有优劣。

还有一些对于手写字符识别问题未来的看法和建议：

探索更深层次、更复杂的网络结构，例如 ResNet、Inception 等，在保持计算效率的同时提高模型性能。

应用更多的数据增强技术和预处理方法，以增加数据集的多样性，提高模型的鲁棒性和泛化能力。

结合迁移学习的方法，利用预训练的模型，在手写字符识别问题上进行 fine-tuning。同时，尝试模型融合等集成学习方法，提高模型性能。

随着技术的发展，探索如注意力机制、图神经网络等新技术在手写字符识别中的应用，以提高模型的表达能力。

总体而言，手写字符识别问题仍有许多待解决的挑战，需要进一步的研究和探索新的方法，以应对不同场景下的需求，并不断提升模型性能。

9.附录

基于 LeNet5 模型进行 MNIST 手写数字集的分类，实验完整代码

```
import tensorflow as tf
from tensorflow.keras import layers, models, datasets
import matplotlib.pyplot as plt

# 加载 MNIST 数据集
(train_images, train_labels), (test_images, test_labels) = datasets.mnist.load_data()
train_images = train_images.reshape((60000, 28, 28, 1)).astype('float32') / 255
test_images = test_images.reshape((10000, 28, 28, 1)).astype('float32') / 255

# 构建 LeNet-5 模型
model = models.Sequential([
    layers.Conv2D(6, kernel_size=(5, 5), strides=(1, 1), activation='relu', input_shape=(28, 28, 1), padding='same'),
    layers.AveragePooling2D(pool_size=(2, 2), strides=(2, 2), padding='valid'),
    layers.Conv2D(16, kernel_size=(5, 5), strides=(1, 1), activation='relu', padding='valid'),
    layers.AveragePooling2D(pool_size=(2, 2), strides=(2, 2), padding='valid'),
    layers.Flatten(),
    layers.Dense(120, activation='relu'),
    layers.Dense(84, activation='relu'),
    layers.Dense(10, activation='softmax')
])

# 编译模型
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

```
# 训练模型
history = model.fit(train_images, train_labels, epochs=10, batch_size=16, validation_data=(test_images, test_labels))

# 可视化训练过程
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0, 1])
plt.legend(loc='lower right')
plt.show()


# 评估模型
test_loss, test_acc = model.evaluate(test_images, test_labels)
print('Test accuracy:', test_acc)

import numpy as np

# 使用训练好的模型对测试集进行预测
predictions = model.predict(test_images)
predicted_labels = np.argmax(predictions, axis=1)

# 找出预测正确和错误的样本
correct_indices = np.where(predicted_labels == test_labels)[0]
incorrect_indices = np.where(predicted_labels != test_labels)[0]

# 输出部分预测错误的样本
plt.figure(figsize=(10, 10))
for i, incorrect_idx in enumerate(incorrect_indices[:25]):
    plt.subplot(5, 5, i + 1)
    plt.imshow(test_images[incorrect_idx].reshape(28, 28), cmap='gray')
    plt.title(f"Predicted: {predicted_labels[incorrect_idx]}, True: {test_labels[incorrect_idx]}")
    plt.axis('off')
plt.tight_layout()
plt.show()

# 输出部分预测正确的样本
```

```
plt.figure(figsize=(10, 10))
for i, correct_idx in enumerate(correct_indices[:25]):
    plt.subplot(5, 5, i + 1)
    plt.imshow(test_images[correct_idx].reshape(28, 28), cmap='gray')
    plt.title(f"Predicted: {predicted_labels[correct_idx]}, True: {test_labels[correct_idx]}")
    plt.axis('off')
plt.tight_layout()
plt.show()


# 参数配置实验
activations = ['tanh', 'relu', 'sigmoid']
learning_rates = [0.01, 0.001, 0.0001]
optimizers = ['sgd', 'adam', 'rmsprop']
batch_sizes = [16, 32, 64]
epochs = 3

results = []

for act in activations:
    for lr in learning_rates:
        for opt in optimizers:
            for batch_size in batch_sizes:
                model = build_lenet5_model(activation=act)
                model.compile(optimizer=opt, loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
                history = model.fit(train_images, train_labels, batch_size=batch_size,
epochs=epochs, validation_data=(test_images, test_labels))
                val_accuracy = history.history['val_accuracy'][-1]  # 获取最终验证准确率
                results.append((act, lr, opt, batch_size, val_accuracy))

import csv


# 将结果保存到 CSV 文件
filename = 'experiment_results.csv'

with open(filename, mode='w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(['Activation', 'Learning Rate', 'Optimizer', 'Batch Size', 'Validation Accuracy'])
    for result in results:

    # 提取结果并进行可视化
    labels = [f'{act} LR:{lr} Opt:{opt} Batch:{batch}' for act, lr, opt, batch, _ in results]
```

```
    accuracy_values = [accuracy for _, _, _, _, accuracy in results]

    # 创建条形图
    plt.figure(figsize=(15, 8))
    plt.bar(labels, accuracy_values, color='skyblue')

    # 添加标题和标签
    plt.title('Validation Accuracy for Different Hyperparameter Combinations and Batch Sizes')
    plt.xlabel('Hyperparameters and Batch Sizes')
    plt.ylabel('Validation Accuracy')
    plt.xticks(rotation=45, ha='right')
    plt.tight_layout()
    plt.show()

    import csv
    import matplotlib.pyplot as plt

    # 读取 CSV 文件数据
    filename = 'experiment_results.csv'

    with open(filename, mode='r') as file:
        reader = csv.reader(file)
        header = next(reader)  # 读取标题行
        data = [row for row in reader]  # 读取数据行

    # 提取数据
    batch_sizes = set(row[3] for row in data)
    activation_types = set(row[0] for row in data)
    learning_rates = set(row[1] for row in data)
    optimizers = set(row[2] for row in data)

    # 准备数据并过滤性能较差的数据点
    batch_size_data = {}
    threshold = 0.5  # 设定阈值
    for batch_size in batch_sizes:
        accuracies = [float(row[4]) for row in data if row[3] == batch_size and float(row[4]) >
threshold]
        batch_size_data[batch_size] = accuracies

    # 绘制折线图
    plt.figure(figsize=(10, 6))
    for batch_size, accuracies in batch_size_data.items():
        # 根据 batch_size 筛选数据行
        filtered_data = [row for row in data if row[3] == batch_size and float(row[4]) >
```

```
threshold]
            # 获取当前参数的值作为标签
            labels = [f'{row[0]}, {row[1]}, {row[2]}' for row in filtered_data]
            plt.plot(labels, accuracies, marker='o', label=f'Batch Size: {batch_size}')

        # 设置图表属性
        plt.title('Model Performance by Different Batch Sizes')
        plt.xlabel('Model Parameters')
        plt.ylabel('Validation Accuracy')
        plt.xticks(rotation=90)  # 旋转 X 轴标签，以防止重叠
        plt.legend()

        # 显示图表
        plt.tight_layout()
        plt.show()
```

Ps：省略了一些重复的代码