机器学习课程报告：作业 2

一、实验简介

该实验任务是使用神经网络进行 MNIST 手写体数字的识别。在神经网络的设计过程中，使用了单隐层神经网络，并使用 relu 作为激活函数，这也是目前使用最广泛、普遍效果最好的激活函数。除此之外，为了提高算法的性能，本实验使用分别使用正则化、滑动平均模型和指数衰减学习率对算法进行了优化，在 MNIST 手写体数据集上精度取得了 98.42%的效果。为了对比随机梯度下降（SGD）的效果，将其与 Adam 进行了对比。因 MNIST 数据集过于简单，所以在本文中将此网络运用到 Fashion-MNIST 数据集中，这个数据集的识别难度比 MNIST 大，已经逐渐在替代 MNIST 数据集，在 Fashion-MNIST 数据集上，算法效果比在 MNIST 数据集上差一些。

编程语言：Python

Python 版本：python 3.6

框架：TensorFlow 1.8.0

二、数据集

本实验使用的数据集是 MNIST 手写体数字识别数据集（http://yann.lecun.com/exdb/mnist/）。 MNIST 数据集包含 60000 张图片作为训练数据，10000 张图片作为测试数据。在 MNIST 中每一个图片都代表了一个 0~9 的数字，下图是其中一个图片的示例：

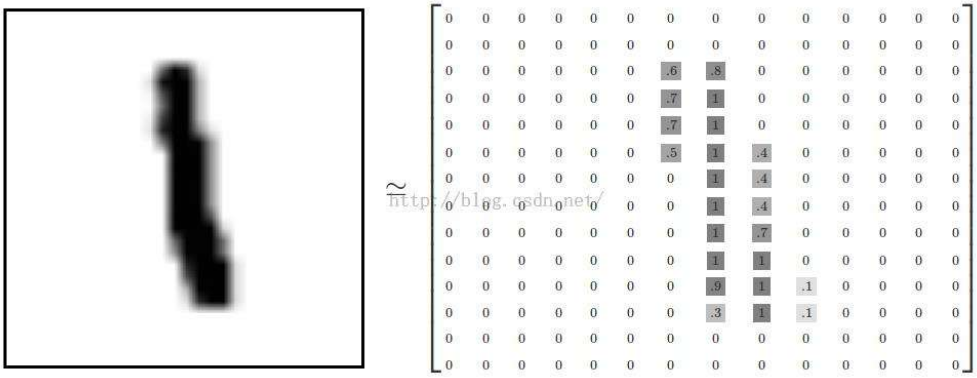

图 1. 数字图片及其像素矩阵

MNIST 数据集中每张图片的大小都是 28*28，图 1 展示了一张图片及其像素矩阵，这张图片表示的是数字“1”。MNIST 数据集总共有 4 个下载文件，如表 1 所示：

文件名称	内容
train-images-idx3-ubyte	训练数据图片
train-labels-idx1-ubyte	训练数据答案
t10k-images-idx3-ubyte	测试数据图片

t10k-labels-idx1-ubyte	测试数据答案

tensorflow 提供了已经封装好的 MNIST 数据，并且通过很简单的代码语句便可以调用，如下所示：

```
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets("./MNIST_data/", one_hot=True) # input_data.read_data_sets 函数生成的类会自动将 MNIST 数据集划分为 train、validation 和 test 三个数据集

print("Training data size: ", mnist.train.num_examples)
print("Validation data size: ", mnist.validation.num_examples)
print("Testing data size：", mnist.test.num_examples)
```

代码执行结果如下：

```
Training data size:  55000
Validation data size:  5000
Testing data size：  10000
```

从代码执行结果可知，train 这个集合内有 55000 张图片，validation 集合内有 5000 张图片，test 集合内有 10000 张图片。处理后的每张图片维度为 784，即 28*28，每一个元素都代表一个像素值。

三、基于 TensorFlow 框架训练神经网络

在 TensorFlow 上使用神经网络解决 MNIST 手写体数字识别的问题。在神经网络的结构上，使用了两层网络结构（一层隐藏层）来实验，并使用激活函数实现神经网络的去线性化，本实验使用 relu 作为神经网络的激活函数，这个函数是目前应用最广泛、效果最好的激活函数。在训练神经网络时，为了使提高算法的准确性，使用了带指数衰减的学习率设置、使用正则化来避免过度拟合，以及使用滑动平均模型来使得最终模型更加健壮。

代码如下：

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data  # 用于读取数据

# 参数设置
INPUT_NODE = 784  # 输入层的节点数。784=28*28
OUTPUT_NODE = 10  # 输出层的节点数，即类别的数目，即 0~9
LAYER1_NODE = 500  # 隐藏层节点数。本实验使用单隐层网络
BATCH_SIZE = 64  # 一个训练 batch 中的训练数据个数。

LEARNING_RATE_BASE = 0.8  # 学习率
LEARNING_RATE_DECAY = 0.99  # 学习率的衰减率
```

```
REGULARIZATION_RATE = 0.0001    # 正则化项在损失函数中的系数
TRAINING_STEPS = 30000    # 训练轮数
MOVING_AVERAGE_DECAY = 0.99    # 滑动平均衰减率，decay 越大模型越趋于稳定


# 计算神经网络的前向传播结果
def forwardPropagation(input, avg_class, weights1, biases1, weights2, biases2):
    if avg_class == None:
        layer1 = tf.nn.relu(tf.matmul(input, weights1) + biases1)
        output = tf.matmul(layer1, weights2) + biases2
        return output
    else:
        layer1 = tf.nn.relu(tf.matmul(input, avg_class.average(weights1)) +
avg_class.average(biases1))
        output = tf.matmul(layer1, avg_class.average(weights2)) +
avg_class.average(biases2)
        return output

# 训练模型的过程
def train(mnist):
    tf.set_random_seed(1)

    x = tf.placeholder(tf.float32, [None, INPUT_NODE], name='x-input')
    y_ = tf.placeholder(tf.float32, [None, OUTPUT_NODE], name='y-input')

    # 参数初始化
    weights1 = tf.Variable(tf.truncated_normal([INPUT_NODE, LAYER1_NODE],
stddev=0.1))
    biases1 = tf.Variable(tf.constant(0.1, shape=[LAYER1_NODE]))
    weights2 = tf.Variable(tf.truncated_normal([LAYER1_NODE, OUTPUT_NODE],
stddev=0.1))
    biases2 = tf.Variable(tf.constant(0.1, shape=[OUTPUT_NODE]))

    # 计算神经网络前向传播的结果。用于计算滑动平均的类为 None
    y = forwardPropagation(x, None, weights1, biases1, weights2, biases2)       # validation:
(5000, 10)

    global_step = tf.Variable(0, trainable=False)
    variable_averages = tf.train.ExponentialMovingAverage(MOVING_AVERAGE_DECAY,
global_step)
    variables_averages_op = variable_averages.apply(tf.trainable_variables())

    # 使用滑动平均之后的输出值，在计算交叉熵时仍然使用 y，在最后输出时使用
average_y
```

```
    average_y = forwardPropagation(x, variable_averages, weights1, biases1, weights2,
biases2)
    # 计算交叉熵作为刻画预测值和真实值之间差距的损失函数,logits 表示隐藏层线性
变换后非归一化后的结果,label 是神经网络的期望输出(其输入格式需要是(batch_size))，
y_是稀疏表示的，需要转化为非系数表示
    # argmax()输出的是每一列最大值所在的数组下标
    cross_entropy = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=y,
labels=tf.argmax(y_, 1))
    # 计算在当前 batch 中所有样例的交叉熵平均值
    cross_entropy_mean = tf.reduce_mean(cross_entropy)

    # 计算 L2 正则化损失函数
    regularizer = tf.contrib.layers.l2_regularizer(REGULARIZATION_RATE)
    # 计算模型的正则化损失，一般只计算神经网络边上权重的正则化损失，而不是用
偏置项
    regularization = regularizer(weights1) + regularizer(weights2)
    # 总损失等于交叉熵损失和正则化损失的和
    loss = cross_entropy_mean + regularization

    # 设置指数衰减的学习率
    learning_rate = tf.train.exponential_decay(
        LEARNING_RATE_BASE,        # 基础学习率
        global_step,       # 当前迭代的轮数
        mnist.train.num_examples / BATCH_SIZE,        # 扫苗完所有的训练数据需要的
迭代次数
        LEARNING_RATE_DECAY,        # 学习率衰减速度
        staircase=True
    )
    train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss,
global_step=global_step)

    # 每过一遍数据通过反向传播来更新神经网络的参数和参数的滑动平均值
    with tf.control_dependencies([train_step, variables_averages_op]):
        train_op = tf.no_op(name='train')

    correct_prediction = tf.equal(tf.argmax(average_y, 1), tf.argmax(y_, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

    # 初始化会话并开始训练过程：
    with tf.Session() as sess:
        tf.global_variables_initializer().run()       # 初始化参数
        validate_feed_dict = {
            x: mnist.validation.images,         # (5000, 784)
            y_: mnist.validation.labels         # (5000, 10)
```

```
        }

        test_feed_dict = {
            x: mnist.test.images,         # (10000, 784)
            y_: mnist.test.labels         # (10000, 10)
        }

        for i in range(TRAINING_STEPS):
            if i % 1000 == 0:
                validate_acc = sess.run(accuracy, feed_dict=validate_feed_dict)
                test_acc = sess.run(accuracy, feed_dict=test_feed_dict)
                print('After %s training steps, accuracy in validation data is %g, accuracy in
test data is %g' % (i, validate_acc, test_acc))

            xs, ys = mnist.train.next_batch(BATCH_SIZE)
            sess.run(train_op, feed_dict={x: xs, y_: ys})

        # 在训练结束之后，在测试数据上检测神经网络模型的最终正确率
        test_acc = sess.run(accuracy, feed_dict=test_feed_dict)
        print('After %s training steps, accuracy in test data is %g' % (TRAINING_STEPS,
test_acc))

if __name__ == '__main__':
    # 下载数据
    mnist = input_data.read_data_sets("./MNIST_data", one_hot=True)
    train(mnist)
```

输出结果如下：

```
After 0 training steps, accuracy in validation data is 0.078, accuracy in test data is 0.0783
After 1000 training steps, accuracy in validation data is 0.9762, accuracy in test data is 0.9746
After 2000 training steps, accuracy in validation data is 0.9806, accuracy in test data is 0.98
After 3000 training steps, accuracy in validation data is 0.983, accuracy in test data is 0.9818
After 4000 training steps, accuracy in validation data is 0.9838, accuracy in test data is 0.9815
After 5000 training steps, accuracy in validation data is 0.9838, accuracy in test data is 0.9829
After 6000 training steps, accuracy in validation data is 0.9858, accuracy in test data is 0.984
After 7000 training steps, accuracy in validation data is 0.9854, accuracy in test data is 0.9827
After 8000 training steps, accuracy in validation data is 0.985, accuracy in test data is 0.9824
After 9000 training steps, accuracy in validation data is 0.9854, accuracy in test data is 0.9837
After 10000 training steps, accuracy in validation data is 0.986, accuracy in test data is 0.9838
After 11000 training steps, accuracy in validation data is 0.9864, accuracy in test data is 0.9834
After 12000 training steps, accuracy in validation data is 0.9852, accuracy in test data is 0.9829
After 13000 training steps, accuracy in validation data is 0.9848, accuracy in test data is 0.984
After 14000 training steps, accuracy in validation data is 0.9854, accuracy in test data is 0.9834
After 15000 training steps, accuracy in validation data is 0.986, accuracy in test data is 0.9842
```

```
After 16000 training steps, accuracy in validation data is 0.986, accuracy in test data is 0.9842
After 17000 training steps, accuracy in validation data is 0.9854, accuracy in test data is 0.983
After 18000 training steps, accuracy in validation data is 0.985, accuracy in test data is 0.9842
After 19000 training steps, accuracy in validation data is 0.986, accuracy in test data is 0.984
After 20000 training steps, accuracy in validation data is 0.9866, accuracy in test data is 0.9847
After 21000 training steps, accuracy in validation data is 0.986, accuracy in test data is 0.9846
After 22000 training steps, accuracy in validation data is 0.9856, accuracy in test data is 0.9846
After 23000 training steps, accuracy in validation data is 0.9854, accuracy in test data is 0.9838
After 24000 training steps, accuracy in validation data is 0.9868, accuracy in test data is 0.9847
After 25000 training steps, accuracy in validation data is 0.9854, accuracy in test data is 0.9845
After 26000 training steps, accuracy in validation data is 0.9858, accuracy in test data is 0.9842
After 27000 training steps, accuracy in validation data is 0.986, accuracy in test data is 0.9847
After 28000 training steps, accuracy in validation data is 0.985, accuracy in test data is 0.985
After 29000 training steps, accuracy in validation data is 0.9854, accuracy in test data is 0.9842
After 30000 training steps, accuracy in test data is 0.9842
```

通过参数调节，在单隐层神经网络中，当学习率为 0.8 时测试集上的正确率达到最高。从结果可以看出，在训练初期，随着训练的进行，验证集上的精度越来越高，直到最后达到稳定状态。最后在测试集上的精度是 98.42%。从执行结果可以看出，此神经网络在 MNIST 手写体数据集识别上的收敛速度非常快，在 5000 步左右在验证集上就达到了 98.38%的效果。

算法损失函数的变化如下图 2 所示（tensorboard 访问浏览器获得），从图中可以看出，随着迭代次数的增加，算法损失不断减小，说明算法不断在学习，直到最后算法收敛。

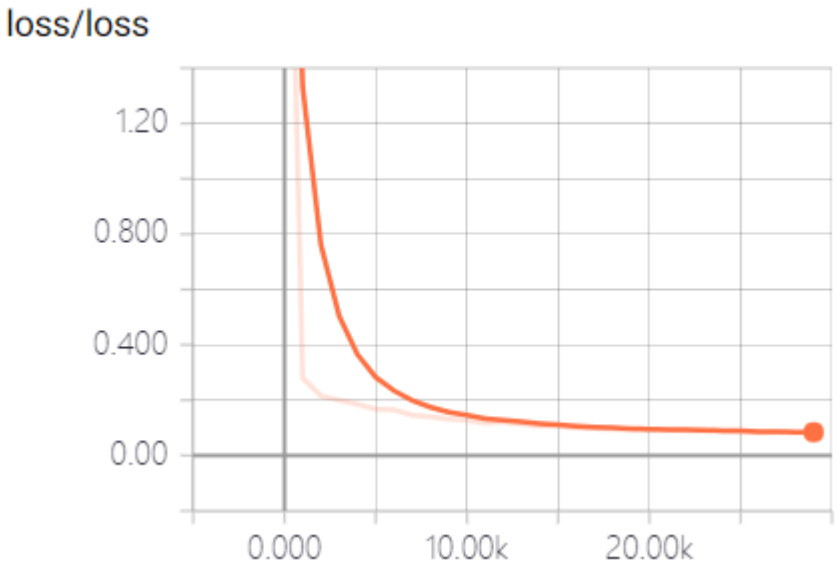

图 2. 损失函数变化图

四、使用验证集判断模型效果

MNIST 数据集使用所有优化方法情况下在验证集和测试集上的精度变化情况如下图 3 所示：

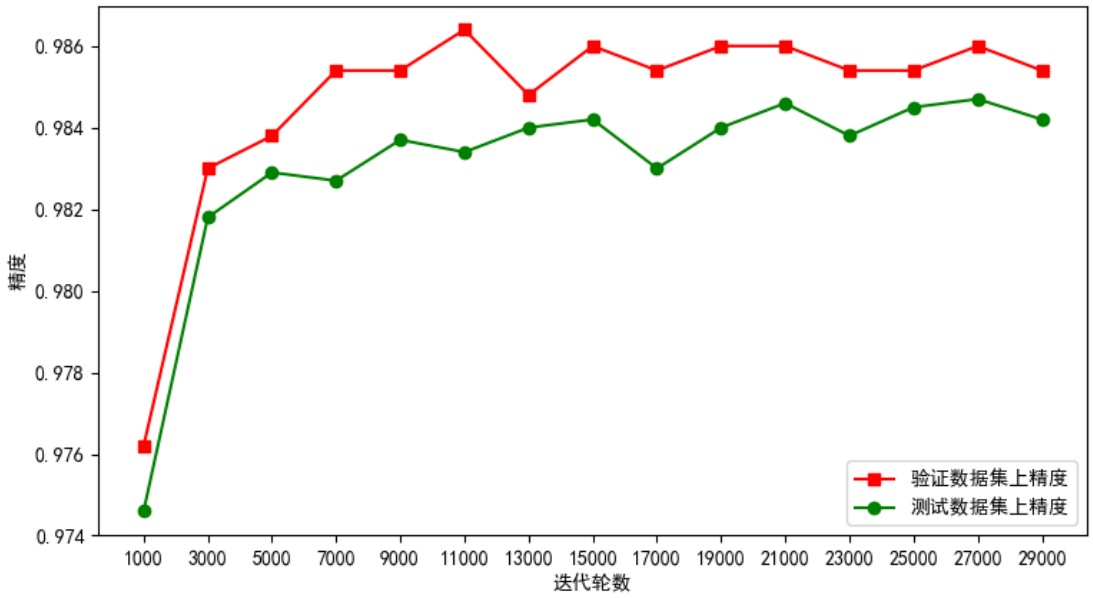

图 3. 精度随迭代次数变化趋势

从图中可以看出，对于 MNIST 数据集，验证集上的精度和测试集上的精度变化趋势是基本保持一致的，说明这个神经网络可以用于手写体的识别。另外，神经网络在 MNIST 数据集上的收敛速度比较快，若在比较复杂的数据集上训练，收敛速度应该会慢一些。

五、不同模型效果比较

在第三节中构建的神经网络使用了 5 种优化方式，并在使用激活函数时使用了效果比较好的 relu 函数，优化算法采用的是梯度下降。5 种优化方法分别是：

- 使用激活函数
- 加入隐藏层
- 使用滑动平均
- 使用正则化
- 使用指数衰减学习率

神经网络在 MNIST 取得了较为不错的效果，模型精度大约在 98.42%左右。为了判断这 5 种优化中的哪些优化方法对于算法结果取得了比较重要的作用，下面就这 5 种优化算法，分析各自对于 MNIST 数据集的精度，模型精度对比如下图 4。为了对比梯度下降（SGD）和 Adam 的效果，在最后又对比了 Adam 算法，因 Adam 算法自适应学习率，所以代码中将指数衰减学习率的优化方法去除。

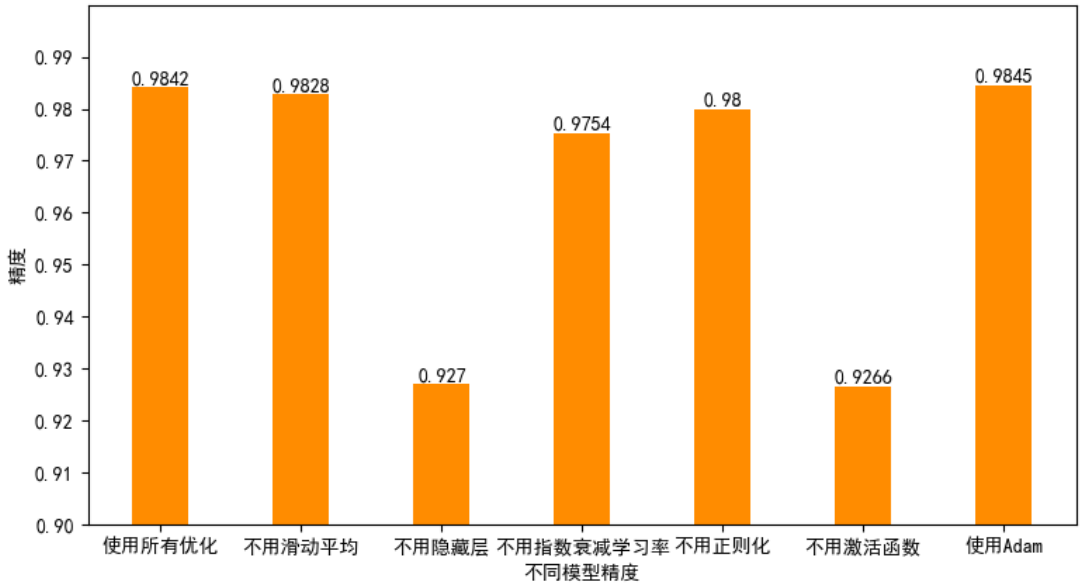

图 4. 不同模型精度对比

从图中可以看出，调整神经网络的结构对最终的正确率有非常大的影响。在没有隐藏层时的精度只能达到 92.7%，不用激活函数的状态下精度只能达到 92.66%。而且不用滑动平均、正则化和指数衰减学习率时的精度和使用所有优化情况下的差别不是很大。这说明神经网络的结构对最终模型的效果有着本质的影响。所以在使用深度学习解决实际任务时，要根据任务的复杂程度设计网络层数，并通过不断调参，确定最优层数。另外，使用 Adam 优化方法的精度达到了 98.45%，比使用 SGD 算法的精度稍高，说明使用 Adam 优化算法的效果更好一些。

六、在 MNIST 数据集和 Fashion-MNIST 数据集精度对比

本文主要基于 MNIST 数据集，但是 MNIST 数据集过于简单，即使是简单地神经网络模型也能达到很好的效果，并且算法很快就可以收敛。Fashion-MNIST 数据集是 MNIST 的替代品，其图片大小、训练样本数、测试样本数和类别数与经典的 MNIST 完全相同。（网址：https://github.com/zalandoresearch/fashion-mnist）

数据集的样子如图 5 所示，其中每个类别占用了三行：

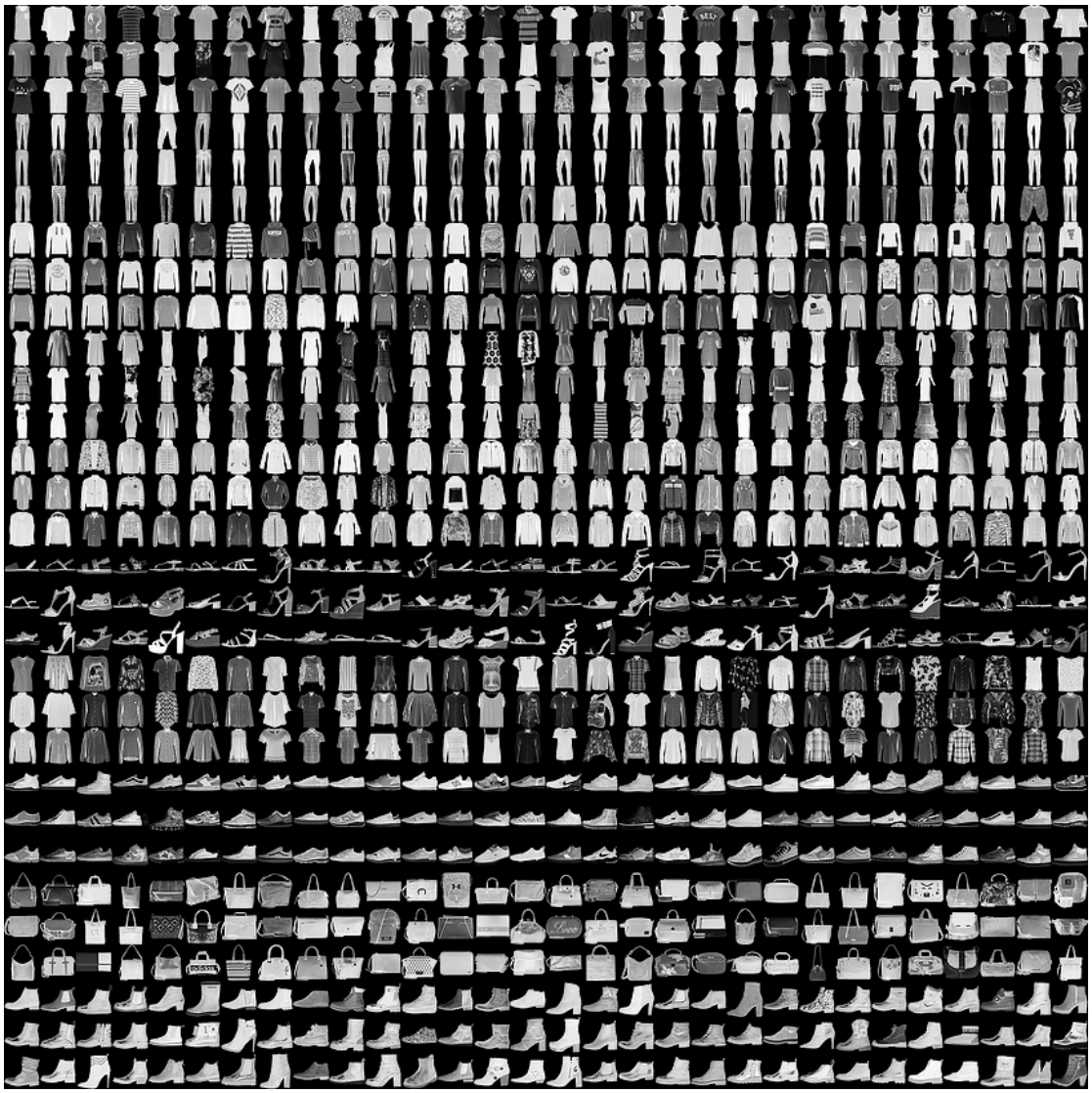

图 5. Fashion-MNIST 数据集样例

其主要标注类别如图 6 所示：

标注编号	描述
0	T-shirt/top（T恤）
1	Trouser（裤子）
2	Pullover（套衫）
3	Dress（裙子）
4	Coat（外套）
5	Sandal（凉鞋）
6	Shirt（汗衫）
7	Sneaker（运动鞋）
8	Bag（包）
9	Ankle boot（踝靴）

图 6. Fashion-MNIST 数据集类别

下图是在 Fashion-MNIST 数据集上数据集的效果（参数设置和在 MNIST 数据集上效果一样）：

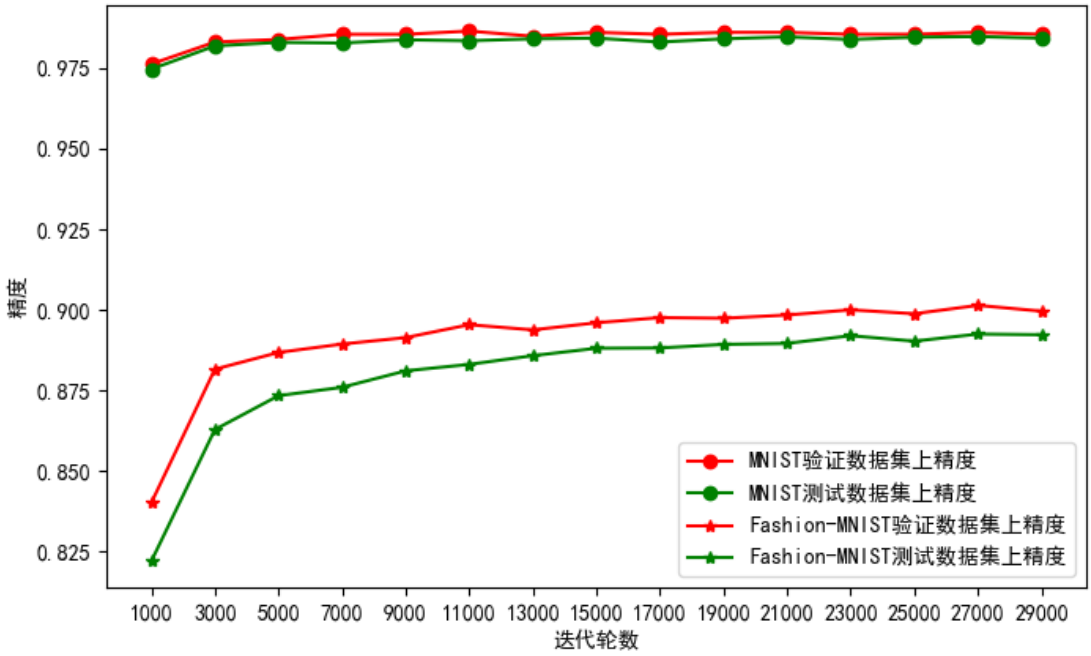

图 7. MNIST 和 Fashion-MNIST 上精度对比

最优情况下，MNIST 数据集在学习率是 0.8 时取得最好的效果，在测试集上的精度为 98.42%；Fashion-MNIST 数据集在学习率是 0.6 时取得最好的效果，在测试机上的精度是 89.22%，因此同样的网络下，算法在较复杂的数据集上的效果要差些。

七、总结

本次实验主要使用神经网络识别 MNIST 手写体数据集，基于 TensorFlow 深度学习框架。在神经网络的构建过程中使用了一层隐层和 relu 激活函数，并使用了正则化、滑动平均以及指数衰减学习率来提高算法的精度。将梯度下降算法（SGD）和 Adam 优化算法进行了对比。最后将此神经网络模型运用到 Fashion-MNIST 数据集中，并与 MNIST 数据集性能进行了对比。

本次实验主要基于神经网络，精度最高达到 98.42%，在使用 Adam 优化算法的情况下，精度达到了 98.45%。若使用卷积神经网络，精度会更高，但因本人关于 CNN 的知识点还未完全理清，所以在本次实验报告中 MNIST 使用 CNN 模型的部分不再详述。