# 18.2 搭建神经网络进行手写字体识别

下面向大家介绍经典的手写字体识别数据集—Mnist数据集，如图18-4所示。数据集中包括0~9十个数字，我们要做的就是对图像进行分类，让神经网络能够区分这些手写字体。

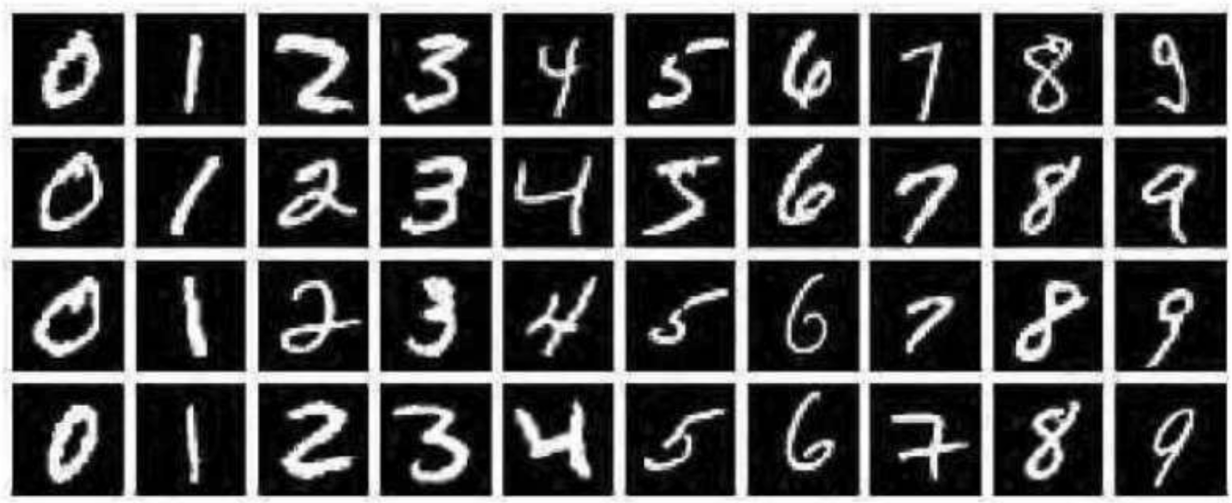


图18-4 Mnist数据集

选择这份数据集的原因是其规模较小(28×28×1)，用笔记本电脑也能执行它，非常适合学习。通常 情况下，数据大小(对图像数据来说，主要是长、宽、大、小)决定模型训练的时间，对于较大的数据 集(例如224×224×3)，即便网络模型简化，还是非常慢。对于没有GPU的初学者来说，在图像处理任务 中，Mnist数据集就是主要练习对象。

## 下载数据集

In [9]:

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

# 使用 TensorFlow 2.x 的 Keras 接口加载 MNIST 数据集
mnist = tf.keras.datasets.mnist


print ("大吉大利 今晚吃鸡")
```

大吉大利 今晚吃鸡

In [11]:

```
import numpy as np
import os
import tensorflow as tf

# 使用 Keras 的接口加载 MNIST 数据集
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# 本地目录路径
data_dir = './my_mnist_data/'
if not os.path.exists(data_dir):
    os.makedirs(data_dir)

# 将数据保存为 .npy 文件
np.save(os.path.join(data_dir, 'x_train.npy'), x_train)
np.save(os.path.join(data_dir, 'y_train.npy'), y_train)
np.save(os.path.join(data_dir, 'x_test.npy'), x_test)
np.save(os.path.join(data_dir, 'y_test.npy'), y_test)

print("MNIST 数据集已保存到本地目录！")
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-k
eras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 30s 3us/step
MNIST 数据集已保存到本地目录!
```

# 查看数据格式

In [12]:

```
import numpy as np
import os

# 本地目录路径
data_dir = './my_mnist_data/'

# 从本地加载数据
x_train = np.load(os.path.join(data_dir, 'x_train.npy'))
y_train = np.load(os.path.join(data_dir, 'y_train.npy'))
x_test = np.load(os.path.join(data_dir, 'x_test.npy'))
y_test = np.load(os.path.join(data_dir, 'y_test.npy'))

# 打印数据信息
print("MNIST 数据集已从本地加载！")
print("训练数据的数量:", x_train.shape[0])
print("测试数据的数量:", x_test.shape[0])
```

```
MNIST 数据集已从本地加载!
训练数据的数量: 60000
测试数据的数量: 10000
```

In [14]:

```
import tensorflow as tf

# 使用 Keras 的接口加载 MNIST 数据集
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# 查看训练集和测试集的基本信息
print("训练集图像数据类型: ", type(x_train))
print("训练集标签数据类型: ", type(y_train))
print("训练集图像的形状: ", x_train.shape)
print("训练集标签的形状: ", y_train.shape)

print("测试集图像的形状: ", x_test.shape)
print("测试集标签的形状: ", y_test.shape)

# 查看其中一张图片的具体数据
print("\n第一张训练图片数据:\n", x_train[0])
print("对应的标签: ", y_train[0])

# 查看数据的最小值和最大值
print("\n训练集图像的最小值: ", x_train.min())
print("训练集图像的最大值: ", x_train.max())
```

```
训练集图像数据类型： <class 'numpy.ndarray'>
训练集标签数据类型： <class 'numpy.ndarray'>
训练集图像的形状： (60000, 28, 28)
训练集标签的形状： (60000,)
测试集图像的形状： (10000, 28, 28)
测试集标签的形状： (10000,)

第一张训练图片数据：
 [[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   3  18  18  18 12
6 136
  175  26 166 255 247 127   0   0   0   0]
 [  0   0   0   0   0   0   0   0  30  36  94 154 170 253 253 253 25
3 253
  225 172 253 242 195  64   0   0   0   0]
 [  0   0   0   0   0   0   0  49 238 253 253 253 253 253 253 253 25
3 251
   93  82  82  56  39   0   0   0   0   0]
 [  0   0   0   0   0   0   0  18 219 253 253 253 253 253 198 182 24
7 241
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0  80 156 107 253 253 205  11   0  4
3 154
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0  14   1 154 253  90   0   0
0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0 139 253 190   2   0
0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0  11 190 253  70   0
0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0  35 241 225 160 10
8   1
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0  81 240 253 25
3 119
   25   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0  45 186 25
3 253
  150  27   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  16  9
3 252
  253 187   0   0   0   0   0   0   0   0]
```

```
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0 249
  253 249  64   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0  46 130 18
3 253
  253 207   2   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0  39 148 229 253 25
3 253
  250 182   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0  24 114 221 253 253 253 25
3 201
   78   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0  23  66 213 253 253 253 253 198   8
1   2
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0  18 171 219 253 253 253 253 195  80   9
0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0  55 172 226 253 253 253 253 244 133  11   0   0
0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0 136 253 253 253 212 135 132  16   0   0   0   0
0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0
    0   0   0   0   0   0   0   0   0   0]]
对应的标签：  5

训练集图像的最小值：  0
训练集图像的最大值：  255
```

**总结：**

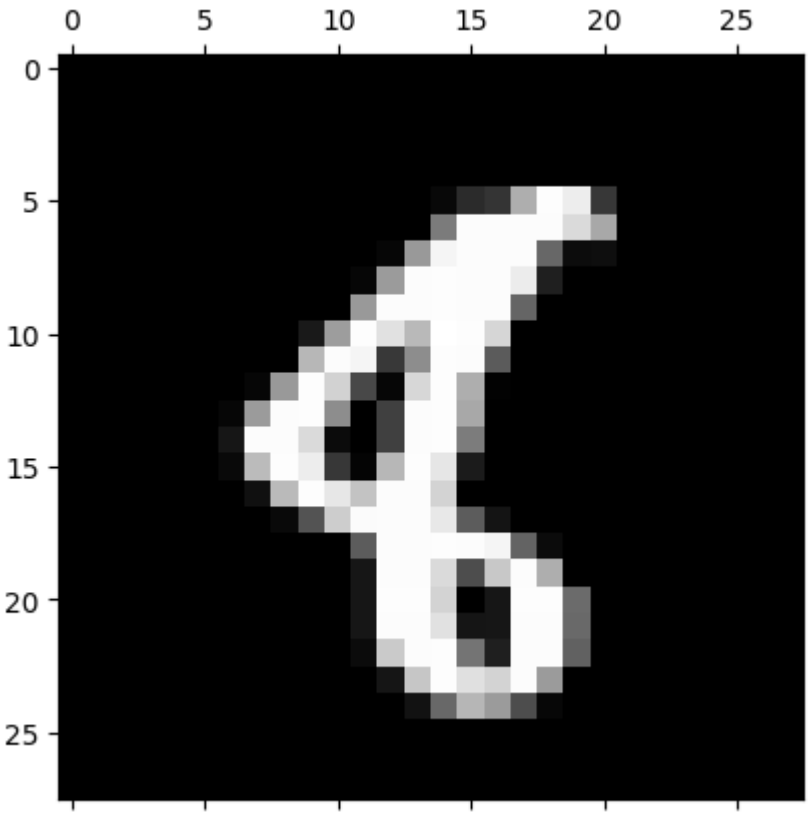
- `x_train` 和 `x_test`：存储图像数据，形状为 `(60000, 28, 28)` 和 `(10000, 28, 28)`，表示有 60,000 张训练图片和 10,000 张测试图片，每张图片大小为 28x28 像素。
- `y_train` 和 `y_test`：存储标签数据，分别为 `(60000,)` 和 `(10000,)`，表示每张图片对应的标签（0 到 9 的数字）。
- **数据值范围**：图像像素值范围是 0 到 255，表示灰度值。

通过这种方式，你可以轻松地查看 MNIST 数据集的基本信息。

In [23]:

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

# 使用 TensorFlow/Keras 加载 MNIST 数据集
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# 设置要展示的样本数量
nsample = 1

# 随机选择要展示的样本索引
randidx = np.random.randint(0, x_train.shape[0], size=nsample)

# 遍历随机选择的样本并展示它们
for i in randidx:
    # 获取当前图像和标签
    curr_img = x_train[i]
    curr_label = y_train[i]

    # 显示图像
    plt.matshow(curr_img, cmap='gray')

    # 打印标签信息
    print(f"{i}th 训练数据, 标签是: {curr_label}")

    # 展示图片
    plt.show()
```

```
46658th 训练数据, 标签是: 8
```



# 构建神经网络模型

接下来就要构造一个神经网络模型来完成手写字体识别，先来梳理一下整体任务流程(见图18-5)。

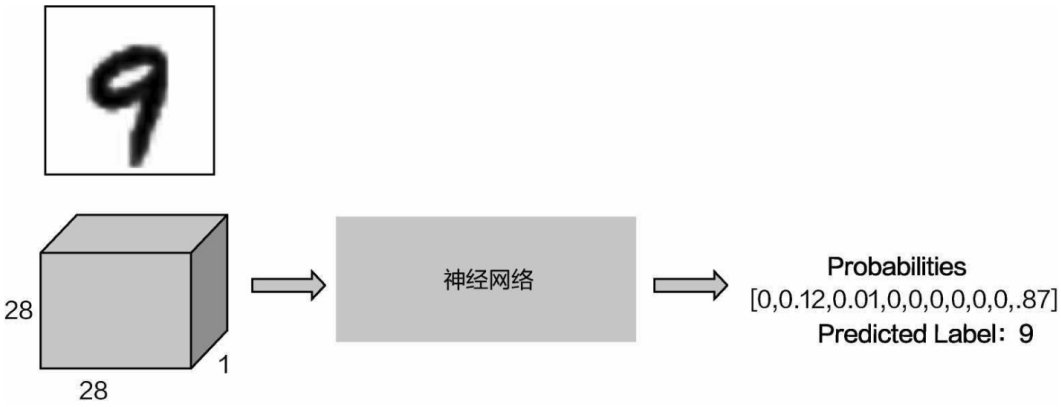



图18-5 神经网络工作流程

通过TensorFlow加载进来的Mnist数据集已经制作成一个个batch数据，所以直接拿过来用就可以。最终的结果就是分类任务，可以得到当前输入属于每一个类别的概率值，需要动手完成的就是中间的网络结构部分。

网络结构定义如图18-6所示，首先定义一个简单的只有一层隐藏层的神经网络，需要两组权重参数分别连接输入数据与隐藏层和隐藏层与输出结果，其中输入数据已经给定784个像素点（28×28×1），输出结果也是固定的10个类别，只需确定隐藏层神经元个数，就可以搭建网络模型。

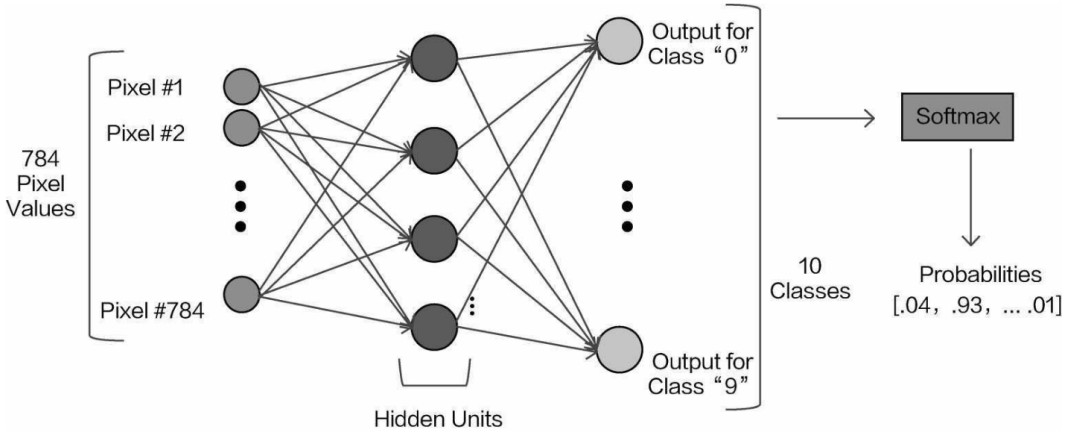



图18-6 网络结构定义

按照任务要求，设置一些网络参数，包括输入数据的规模、输出结果规模、隐藏层神经元个数以及迭代次数与batchsize大小：

In
```
numClasses = 10
inputSize = 784
numHiddenUnits = 64
trainingIterations = 10000
batchSize = 64
```

numClasses固定成10，表示所有数据都是用于完成这个十分类任务。隐藏层神经元个数可以自由设置，在实际操作过程中，大家也可以动手调节其大小，以观察结果的变化，对于Mnist数据集来说，64个就足够了。

In
```
X = tf.placeholder(tf.float32, shape = [None, inputSize])
y = tf.placeholder(tf.float32, shape = [None, numClasses])
```

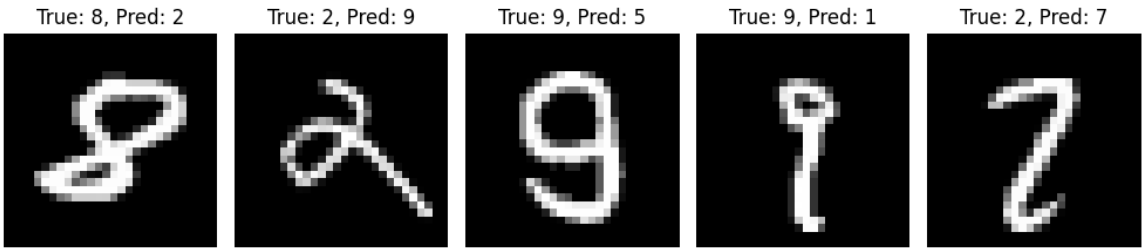
## 一层隐藏层效果

In [45]:

```
import tensorflow as tf
from tensorflow.keras import layers, models
import numpy as np

# 1. 加载MNIST数据集
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# 数据预处理：将图像展平为784维向量，并将像素值归一化到[0, 1]
x_train = x_train.reshape(-1, 28*28).astype('float32') / 255.0
x_test = x_test.reshape(-1, 28*28).astype('float32') / 255.0

# 2. 构建简单的神经网络模型
model = models.Sequential([
    layers.InputLayer(input_shape=(28*28,)),  # 输入层，28x28像素展平为784个输入
    layers.Dense(64, activation='relu'),      # 隐藏层，64个神经元，激活函数为ReLU
    layers.Dense(10, activation='softmax')    # 输出层，10个类别，使用Softmax激活
])

# 3. 编译模型，指定损失函数、优化器和评估指标
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# 4. 训练模型
model.fit(x_train, y_train, epochs=20, batch_size=32)

# 5. 在测试集上评估模型
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"测试集准确率: {test_acc:.4f}")

# 6. 进行所有测试集的预测
predictions = model.predict(x_test)

# 将预测结果转换为类别索引
predicted_labels = np.argmax(predictions, axis=1)

# 7. 找到预测错误的样本
incorrect_indices = np.where(predicted_labels != y_test)[0]

# 打印前5个预测错误的样本
print(f"找到{len(incorrect_indices)}个预测错误的样本。")

# 8. 可视化前几个预测错误的样本
num_to_display = 5  # 要展示的错误样本数量

plt.figure(figsize=(10, 5))

for i, idx in enumerate(incorrect_indices[:num_to_display]):
    plt.subplot(1, num_to_display, i + 1)
    plt.imshow(x_test[idx].reshape(28, 28), cmap='gray')
    plt.title(f"True: {y_test[idx]}, Pred: {predicted_labels[idx]}")
    plt.axis('off')

plt.tight_layout()
plt.show()
```

```
Epoch 1/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 921us/step - accurac
y: 0.8585 - loss: 0.5115
Epoch 2/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 897us/step - accurac
y: 0.9527 - loss: 0.1604
Epoch 3/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 873us/step - accurac
y: 0.9682 - loss: 0.1095
Epoch 4/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 962us/step - accurac
y: 0.9758 - loss: 0.0840
Epoch 5/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 930us/step - accurac
y: 0.9781 - loss: 0.0690
Epoch 6/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 876us/step - accurac
y: 0.9827 - loss: 0.0583
Epoch 7/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 888us/step - accurac
y: 0.9847 - loss: 0.0514
Epoch 8/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 872us/step - accurac
y: 0.9881 - loss: 0.0403
Epoch 9/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 897us/step - accurac
y: 0.9901 - loss: 0.0340
Epoch 10/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 893us/step - accurac
y: 0.9914 - loss: 0.0302
Epoch 11/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 886us/step - accurac
y: 0.9915 - loss: 0.0269
Epoch 12/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 897us/step - accurac
y: 0.9934 - loss: 0.0231
Epoch 13/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 898us/step - accurac
y: 0.9947 - loss: 0.0196
Epoch 14/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 889us/step - accurac
y: 0.9954 - loss: 0.0169
Epoch 15/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 940us/step - accurac
y: 0.9959 - loss: 0.0145
Epoch 16/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 885us/step - accurac
y: 0.9958 - loss: 0.0140
Epoch 17/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 925us/step - accurac
y: 0.9966 - loss: 0.0132
Epoch 18/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 907us/step - accurac
y: 0.9965 - loss: 0.0115
Epoch 19/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 904us/step - accurac
y: 0.9969 - loss: 0.0102
Epoch 20/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 884us/step - accurac
```

```
y: 0.9971 - loss: 0.0090
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 643us/step - accuracy:
0.9720 - loss: 0.1211
测试集准确率: 0.9746
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 705us/step
找到254个预测错误的样本。
```



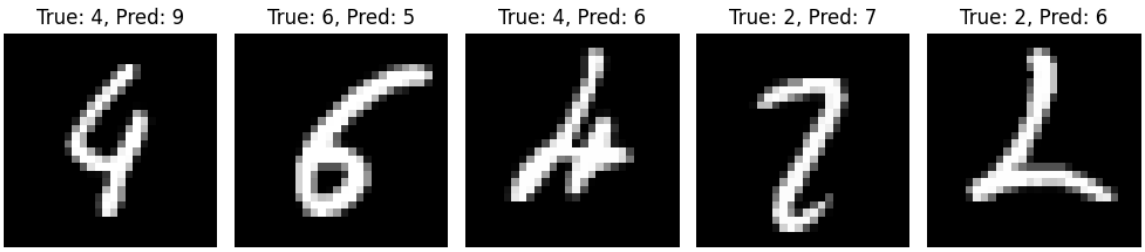
## 两层隐藏层效果

In [48]:

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

# 1. 加载MNIST数据集
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# 数据预处理：将图像展平为784维向量，并将像素值归一化到[0, 1]
x_train = x_train.reshape(-1, 28*28).astype('float32') / 255.0
x_test = x_test.reshape(-1, 28*28).astype('float32') / 255.0

# 2. 构建具有两层隐藏层的神经网络模型
model = models.Sequential([
    layers.InputLayer(input_shape=(28*28,)),  # 输入层，28x28像素展平为784个输入
    layers.Dense(64, activation='relu'),      # 第一层隐藏层，64个神经元，ReLU激活
    layers.Dense(1024, activation='relu'),       # 第二层隐藏层，128个神经元，ReLU激活
    layers.Dense(10, activation='softmax')    # 输出层，10个类别，使用Softmax激活
])

# 3. 编译模型，指定损失函数、优化器和评估指标
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# 4. 训练模型
model.fit(x_train, y_train, epochs=20, batch_size=32)

# 5. 在测试集上评估模型
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"测试集准确率: {test_acc:.4f}")

# 6. 进行所有测试集的预测
predictions = model.predict(x_test)

# 将预测结果转换为类别索引
predicted_labels = np.argmax(predictions, axis=1)

# 7. 找到预测错误的样本
incorrect_indices = np.where(predicted_labels != y_test)[0]

# 打印前5个预测错误的样本
print(f"找到{len(incorrect_indices)}个预测错误的样本。")

# 8. 可视化前几个预测错误的样本
num_to_display = 5  # 要展示的错误样本数量

plt.figure(figsize=(10, 5))

for i, idx in enumerate(incorrect_indices[:num_to_display]):
    plt.subplot(1, num_to_display, i + 1)
    plt.imshow(x_test[idx].reshape(28, 28), cmap='gray')
    plt.title(f"True: {y_test[idx]}, Pred: {predicted_labels[idx]}")
    plt.axis('off')

plt.tight_layout()
plt.show()
```

```
Epoch 1/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy:
0.8772 - loss: 0.4099
Epoch 2/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step - accuracy:
0.9671 - loss: 0.1090
Epoch 3/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy:
0.9764 - loss: 0.0748
Epoch 4/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy:
0.9823 - loss: 0.0568
Epoch 5/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step - accuracy:
0.9854 - loss: 0.0462
Epoch 6/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step - accuracy:
0.9874 - loss: 0.0370
Epoch 7/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy:
0.9906 - loss: 0.0286
Epoch 8/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step - accuracy:
0.9903 - loss: 0.0293
Epoch 9/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy:
0.9919 - loss: 0.0248
Epoch 10/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy:
0.9932 - loss: 0.0188
Epoch 11/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy:
0.9935 - loss: 0.0190
Epoch 12/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy:
0.9947 - loss: 0.0158
Epoch 13/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy:
0.9950 - loss: 0.0158
Epoch 14/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy:
0.9958 - loss: 0.0123
Epoch 15/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy:
0.9950 - loss: 0.0184
Epoch 16/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy:
0.9953 - loss: 0.0146
Epoch 17/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy:
0.9952 - loss: 0.0152
Epoch 18/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy:
0.9962 - loss: 0.0123
Epoch 19/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy:
0.9969 - loss: 0.0103
Epoch 20/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy:
```

```
0.9958 - loss: 0.0148
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.
9740 - loss: 0.1604
测试集准确率: 0.9786
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
找到214个预测错误的样本。
```




# 本章总结

本章选择TensorFlow框架来搭建神经网络模型，初次使用可能会觉得有一些麻烦，但习惯了就会觉得每一步流程都很规范。无论什么任务，核心都在于选择合适的目标函数与输入格式，网络模型和迭代优化通常都是差不多的。大家在学习过程中，还可以选择Cifar数据集来尝试分类任务，同样都是小规模（32×32×3）数据，非常适合练手（见图18-7）。

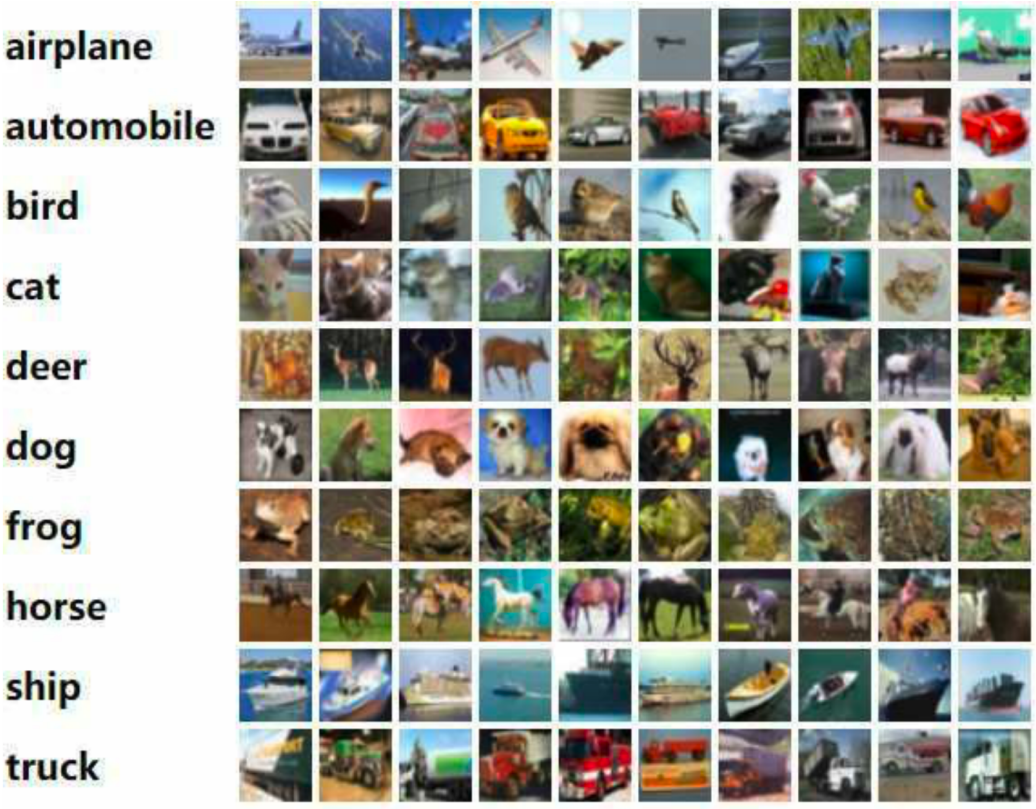



图18-7 Cifar-10数据集