深 圳 大 学 实 验 报 告

课程名称 机器学习

项目名称 实验六：深度神经网络

学　　院 计算机与软件学院

专　　业 软件工程（腾班）

指导教师 赖志辉

报 告 人 黄亮铭 学号 2022155028

实验时间 2024 年 5 月 13 日至 2024 年 6 月 9 日

实验报告提交时间 2024 年 6 月 9 日

教务处制

一、实验目的与要求

1. 了解 CNN 与全连接网络的原理以及它们之间的异同。
2. 了解神经网络前向后向传播算法的优化迭代公式。
3. 熟练使用一种深度神经网络算法，并在人脸识别等领域可以应用。
4. 了解 Transformer，并与上文提到的深度神经网络算法在相应的领域中进行性能测试。

二、实验内容与方法

1. 全方面比较 CNN 与全连接网络的异同。
2. 推导神经网络前向后向传播算法的优化迭代公式。
3. 熟练掌握一种深度神经网络的算法与应用，并给出在人脸识别、身份证识别、通用手写体识别等方面的 2 个以上应用实验案例与效果；介绍 transformer 并与它们（1 中的方法）比较识别性能。

三、实验步骤与过程

1. 全方面比较 CNN 与全连接网络的异同。

   相同点：**①结构相似。**两种神经网络都具有多层结构，每一层由一系列节点组织形成。每一个节点代表一个神经元，不同层之间的节点存在连接的边。**②流程相似。**全连接神经网络的损失函数以及参数的优化过程都适用于 CNN。CNN 的输入输出以及训练的流程和全连接神经网络也基本一致。以通用手写体识别为例为列， CNN 的输入层就是手写体的原始图像，而输出层中的每一个节点代表了不同类别的可信度。这与全连接神经网络的输入输出是一致的。

   不同点：**①相邻层连接方式不同。**CNN 采取局部感知和权值共享的策略。CNN 的核心是卷积层，该层的滤波器（或称卷积核）可以在输入数据上滑动，每次只关注一小块区域，这体现了局部感知特性。同时，一个滤波器在整个输入数据上应用相同的权重，即权值共享，减少了参数数量，提高了模型效率。而全连接网络是完全互连的结构，采取独立权值的策略。全连接网络中的当前层的每个神经元与下一层的所有神经元都有连接，每个连接都有独立的学习参数。**②结构设计思路不同。**CNN 的设计思路为**多层结构**：通常包括卷积层、池化层、激活函数层，最后可能连接几个全连接层进行分类或回归。而全连接网络无特定结构设计。这使得全连接网络更加通用，但是随

之而来的是数量巨大的参数。**③应用场景不同。**CNN 通常用于处理计算机视觉、自然语言处理等任务，而全连接网络通常用于处理传统的机器学习任务和小规模数据集的任务（参数较少全连接网络更加合适，可以避免 CNN 过拟合的风险）。

2. 推导神经网络前向后向传播算法的优化迭代公式。

前提。

输入层一共 d 个参数，

$$[x_1, x_2, \dots\dots, x_d]$$

输出层，一共 l 个参数，

$$[y_1, y_2, \dots\dots, y_l]$$

隐层神经元一共 q 个，输入阈值为γ_h，隐层神经输入为，

$$\alpha_h = \sum_{i=1}^{d} v_{ih} x_i$$

输出神经元输入阈值为θ_j，输出神经元输入为

$$\beta_j = \sum_{h=1}^{q} v_{hj} b_h$$

输入层到隐层的连接边权值分别为

$$[v_{1h}, v_{2h}, \dots\dots, v_{dh}]$$

隐层到输出层的连接边权值分别为

$$[w_{1j}, w_{2j}, \dots\dots, w_{qj}]$$

不妨假设该神经网络使用的激活函数为 Sigmoid 函数。

假定神经网络的输出为

$$\hat{y_j} = f(\beta_j - \theta_j)$$

则网络在(x_k, y_k)上的均方误差为

$$E_k = \frac{1}{2}\sum_{j=1}^{l} (\widehat{y_j^k} - y_j^k)^2$$

前向传播。

如果我们此时输入为

$$[x_1, x_2, \dots\dots, x_d]$$

经过隐层神经元的输出为

$$\left[f\left(\sum_{i=1}^{d} v_{i1}x_i - \gamma_1\right), f\left(\sum_{i=1}^{d} v_{i2}x_i - \gamma_2\right), \dots\dots, f\left(\sum_{i=1}^{d} v_{iq}x_i - \gamma_q\right)\right]$$

最后输出层的输出为

$$\left[f(\sum_{h=1}^{q} w_{h1} f\left(\sum_{i=1}^{d} v_{i1}x_i - \gamma_1\right) - \theta_1), f(\sum_{h=1}^{q} w_{h1} f\left(\sum_{i=1}^{d} v_{i2}x_i - \gamma_2\right)\right)$$

$$- \theta_2), \dots\dots, f(\sum_{h=1}^{q} w_{hl} f\left(\sum_{i=1}^{d} v_{il}x_i - \gamma_l\right) - \theta_l)\Bigg]$$

也就是说，

$$y_l = f(\sum_{h=1}^{q} w_{hl} f\left(\sum_{i=1}^{d} v_{il}x_i - \gamma_l\right) - \theta_l)$$

由此，我们得到了神经网络的前向推导。

后向传播。

对于前式的误差E_k，给定学习率η，有

$$\Delta w_{hj} = -\eta \frac{\partial E_k}{\partial w_{hj}}$$

根据w_{hj}的影响顺序，有

$$\frac{\partial E_k}{\partial w_{hj}} = \frac{\partial E_k}{\partial \widehat{y_j^k}} \frac{\partial \widehat{y_j^k}}{\partial \beta_j} \frac{\partial \beta_j}{\partial w_{hj}}$$

根据β_j的定义，显然有

$$\frac{\partial \beta_j}{\partial w_{hj}} = b_h$$

Sigmoid 函数有一个性质

$$f'(x) = f(x)(1 - f(x))$$

于是根据上面的推导，我们可以得到

$$g_j = -\frac{\partial E_k}{\partial \widehat{y_j^k}} \frac{\partial \widehat{y_j^k}}{\partial \beta_j}$$

$$= -\left(\widehat{y_j^k} - y_j^k\right) f'\left(\beta_j - \theta_j\right)$$

$$= \widehat{y_j^k}\left(1 - \widehat{y_j^k}\right)\left(y_j^k - \widehat{y_j^k}\right)$$

将得到的结果代入最先提出的梯度中，得到

$$\Delta w_{hj} = \eta g_j b_h$$

类似地，我们可以得到

$$\Delta \theta_j = \eta g_j b_h$$

$$\Delta w_{hj} = -\eta g_j$$

$$\Delta v_{ih} = \eta e_h x_i$$

$$\Delta \gamma_h = -\eta e_h$$

其中上式的e_h可以被表示为

$$e_h = -\frac{\partial E_k}{\partial b_h} \frac{\partial b_h}{\partial \alpha_h}$$

$$= -\sum_{j=1}^{l} \frac{\partial E_k}{\partial \beta_j} \frac{\partial \beta_j}{\partial b_h} f'(\alpha_h - \gamma_h)$$

$$= \sum_{j=1}^{l} w_{hj} g_j f'(\alpha_h - \gamma_h)$$

$$= b_h(1 - b_h) \sum_{j=1}^{l} w_{hj} g_j$$

通过上述推导，我们得到了神经网络的更新公式。

综上所述，我成功推导了神经网络前向后向传播算法的优化迭代公式

3. 熟练掌握一种深度神经网络的算法与应用，并给出在人脸识别、身份证识别、通用手写体识别等方面的 2 个以上应用实验案例与效果；介绍 transformer 并与它们（1 中的方法）比较识别性能。

人脸识别应用。

首先，我们需要对数据进行预处理，具体步骤如下。

1. 导入所需要的库。

```
import numpy
import pandas
from PIL import Image
from keras import backend as K
from keras.utils import np_utils
```

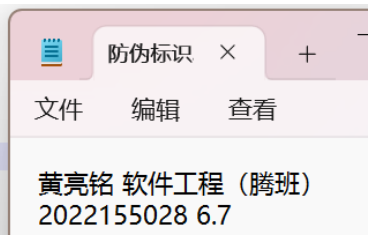

图 1：导入相应库

2. 加载和处理图像数据。

```
def load_data(dataset_path):
    img = Image.open(dataset_path)
    img_ndarray = numpy.asarray(img, dtype='float64') / 256
    print(img_ndarray.shape)
    faces = numpy.empty((400, 57, 47))
    for row in range(20):
        for column in range(20):
            faces[row * 20 + column] = img_ndarray[row * 57:(row + 1) * 57, column * 47:(column + 1) * 47]
    label = numpy.empty(400)
    for i in range(40):
        label[i * 10:i * 10 + 10] = i
    label = label.astype(numpy.int)
    label = np_utils.to_categorical(label, 40)
```

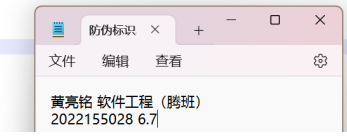

图 2：加载和处理图像的函数

3. 数据集划分。

```
train_data = numpy.empty((320, 57, 47))
train_label = numpy.empty((320, 40))
valid_data = numpy.empty((40, 57, 47))
valid_label = numpy.empty((40, 40))
test_data = numpy.empty((40, 57, 47))
test_label = numpy.empty((40, 40))

for i in range(40):
    train_data[i * 8:i * 8 + 8] = faces[i * 10:i * 10 + 8]
    train_label[i * 8:i * 8 + 8] = label[i * 10:i * 10 + 8]
    valid_data[i] = faces[i * 10 + 8]
    valid_label[i] = label[i * 10 + 8]
    test_data[i] = faces[i * 10 + 9]
    test_label[i] = label[i * 10 + 9]

return [(train_data, train_label), (valid_data, valid_label), (test_data, test_label)]
```

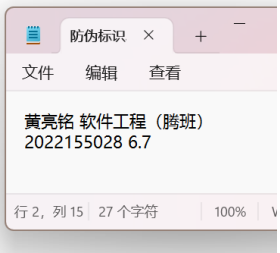

图 3：数据集划分

其次，我们使用一个基于 Keras 框架的 CNN 模型实现人脸识别。具体步骤如下。

1. 导入所需要的库。

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation, Flatten, Conv2D, AveragePooling2D
from PIL import Image
import FaceData
```

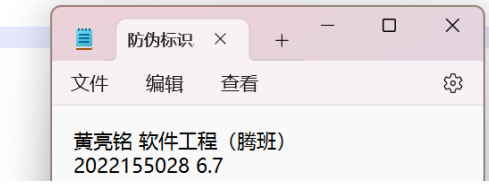

图 4：导入相应的库

2. 加载和预处理数据。

```
[(X_train, Y_train), (X_valid, Y_valid), (X_test, Y_test)] = FaceData.load_data('faces.gif')

X_train = X_train[:, :, :, np.newaxis]
X_valid = X_valid[:, :, :, np.newaxis]
X_test = X_test[:, :, :, np.newaxis]

print('样本数据集的维度：', X_train.shape, Y_train.shape)
print('测试数据集的维度：', X_test.shape, Y_test.shape)
```

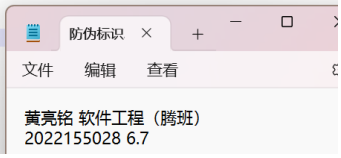

图 5：预处理数据

3. 构建模型。

```
model = Sequential()
model.add(Conv2D(6, kernel_size, input_shape=input_shape, strides=1))
model.add(AveragePooling2D(pool_size=pool_size, strides=2))
model.add(Conv2D(12, kernel_size, strides=1))
model.add(AveragePooling2D(pool_size=pool_size, strides=2))
model.add(Flatten())
model.add(Dense(nb_classes))
model.add(Activation('sigmoid'))
```

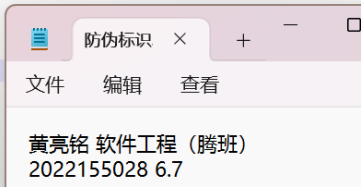

图 6：构建模型

4. 训练模型。

```
model.compile(loss='categorical_crossentropy', optimizer='adadelta', metrics=['accuracy'])
model.fit(X_train, Y_train, batch_size=batch_size, epochs=epochs, verbose=1, validation_data=(X_valid, Y_valid
```

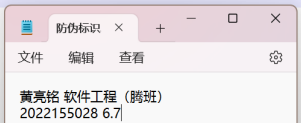

图 7：训练模型

5. 使用模型评估和预测。

```
score = model.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])
y_pred = model.predict(X_test)
y_pred = y_pred.argmax(axis=1)
for i in range(len(y_pred)):
    oneimg = X_test[i, :, :, 0] * 256
    im = Image.fromarray(oneimg)
    print('%d %d' % (i, y_pred[i]))
```

防伪标识 × +
文件 编辑 查看
黄亮铭 软件工程（腾班）
2022155028 6.7

图 8：评估和预测

最后，测试结果如下图。

	10	100	1000	5000	10000	15000	20000
人脸识别率	0.034	0.042	0.053	0.571	0.749	0.948	0.988
lossrate	3.649	3.586	3.528	2.726	1.37	0.508	0.212

黄亮铭 软件工程（腾班）
2022155028 6.7

图 9：模型测试结果

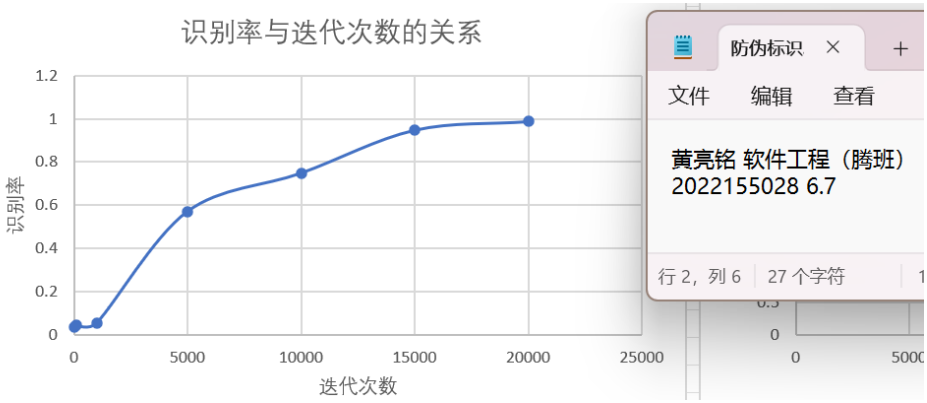

图 10a：模型测试可视化结果

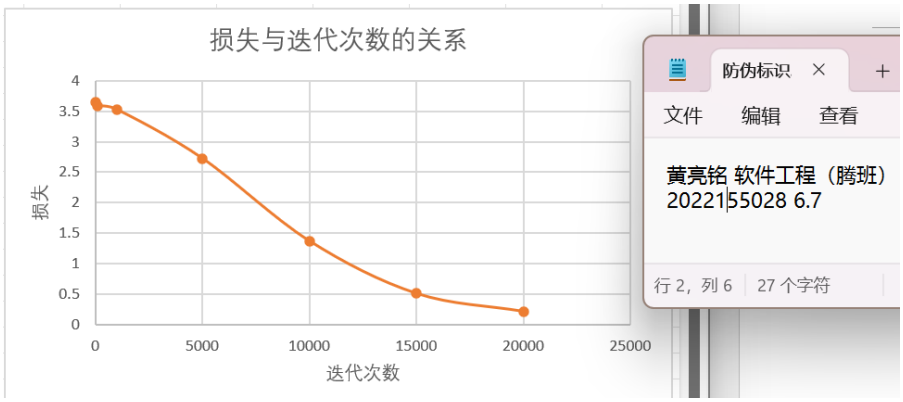

图 10b：模型测试可视化结果

手写体识别应用。

我们使用 BP 算法实现手写识别体识别，具体步骤如下。

1. 导入所需的库。

```
import numpy as np
from sklearn.datasets import load_digits
from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
import matplotlib.pyplot as plt
```

防伪标识 × +
文件 编辑 查看
黄亮铭 软件工程（腾班）
2022155028 6.7

图 11：导入相应库

2. 加载数据集。

```
digits = load_digits()
for i in range(min(digits.images.shape[0], 2)):
    plt.imshow(digits.images[i], cmap='gray')
```

文件 编辑 查看
黄亮铭 软件工程（腾班）
2022155028 6.7

图 12：加载数据集

3. 预处理数据及其标签。

```
X = digits.data
y = digits.target
V = np.random.random((64, 100)) * 2 - 1
W = np.random.random((100, 10)) * 2 - 1
X_train, X_test, y_train, y_test = train_test_split(X, y)
labels_train = LabelBinarizer().fit_transform(y_train)
```

文件 编辑 查看
黄亮铭 软件工程（腾班）
2022155028 6.7

图 13：预处理

4. 定义相关函数。

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def dsigmoid(x):
    return x * (1 - x)

def train(X, y, steps=10000, lr=0.011):
    global V, W
    for n in range(steps + 1):
        i = np.random.randint(X.shape[0])
        x = X[i]
        x = np.atleast_2d(x)
        L1 = sigmoid(np.dot(x, V))
        L2 = sigmoid(np.dot(L1, W))
        L2_delta = (y[i] - L2) * dsigmoid(L2)
        L1_delta = L2_delta.dot(W.T) * dsigmoid(L1)
        W += lr * L1.T.dot(L2_delta)
        V += lr * x.T.dot(L1_delta)

        if n % 1000 == 0:
            output = predict(X_test)
            predictions = np.argmax(output, axis=1)
            acc = np.mean(np.equal(predictions, y_test))
            dW = L1.T.dot(L2_delta)
            dV = x.T.dot(L1_delta)
            gradient = np.sum([np.sqrt(np.sum(np.square(j))) for j in [dW, dV]])
            print('steps', n, 'accuracy', acc, 'gradient', gradient)

def predict(x):
    L1 = sigmoid(np.dot(x, V))
    L2 = sigmoid(np.dot(L1, W))
    return L2
```

防伪标识 × +
文件 编辑 查看
黄亮铭 软件工程（腾班）
2022155028 6.7
行 2，列 15 27 个字符 100%

图 14：相关函数

5. 训练和评估模型。

```
train(X_train, labels_train, 100000, lr=0.11)
train(X_train, labels_train, 100000, lr=0.011)

output = predict(X_test)
predictions = np.argmax(output, axis=1)
acc = np.mean(np.equal(predictions, y_test))
print('accuracy', acc)
print(classification_report(predictions, y_test, digits=4))
```

防伪标识 ×

文件 编辑 查看

黄亮铭 软件工程（腾班）
2022155028 6.7

图 15：训练并评估模型

评估结果如下图所示。

	10	100	1000	5000	10000	15000	20000
人脸识别率	0.088	0.13	0.482	0.869	0.927	0.956	0.962

黄亮铭 软件工程（腾班）
2022155028 6.7

图 16：评估结果

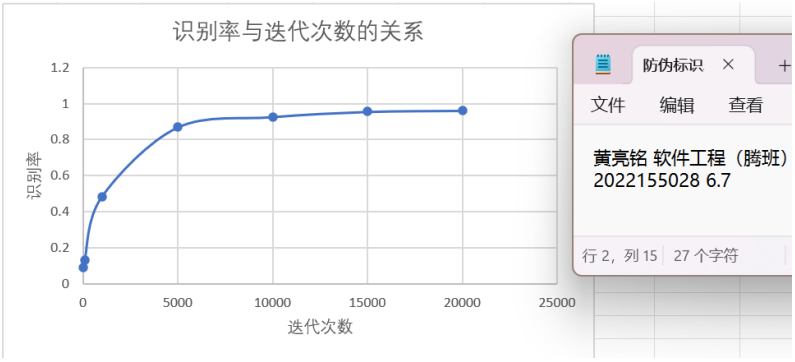

图 17：可视化结果

Transformer 介绍。

Transformer 是一种用于自然语言处理（NLP）和其他任务的神经网络架构，由 Vaswani 等人在 2017 年的论文《Attention Is All You Need》中提出。Transformer 模型因其强大的性能和并行处理能力，已经成为许多 NLP 任务的主流选择，如机器翻译、文本生成、文本分类等。

Transformer 有以下几个核心概念：

1. **自注意力机制。**Transformer 的核心是自注意力机制，它允许模型在处理输入序列的每个位置时，动态地关注序列中的其他位置。这种机制使模型能够捕捉长距离的依赖关系，而不受序列长度的限制.
2. **多头注意力机制。**通过使用多头注意力机制，模型能够并行地从不同的子空间中提取信息，增强了模型的表达能力。每个注意力头独立地学习不同的表示，然后将它们组合起来。
3. **位置编码。**由于 Transformer 模型没有循环结构，需要通过位置编码来注入序列中每个位置的信息。这些编码被加到输入的词嵌入中，使模型能够区分不同位置的词。

4. **编码器-解码器结构。**Transformer 由编码器和解码器两个主要部分组成。编码器将输入序列转换为一系列隐藏状态，解码器则根据这些隐藏状态生成输出序列。编码器和解码器都由多个相同的层堆叠而成，每层包含多头自注意力机制和前馈神经网络。

Transformer 的主要结构为编码器和解码器。编码器由子层连接，分别为多头自注意力机制和前馈神经网络。每个子层后都存在一个层归一化和残差连接。解码器与编码器类似，但是解码器有三个子层，分别为一个多头自注意力机制、一个与编码器的多头自注意力机制和一个前馈神经网络。解码器的自注意力机制被修改为“掩码”子注意力，以确保当前时间帧只能关注先前的时间帧，防止信息泄露。

Transformer 存在以下几点优势：①并行化处理；②擅长捕捉长距离的依赖关系；③可扩展性强大。

Transformer 的主要应用场景：①机器翻译；②文本生成；③文本摘要；④问答机器人。Transformer 的应用场景非常广阔，在多个领域均展现出非常强大的潜力，是目前主流的神经网络框架。

Transformer 实现手写体识别，并于前面的提及的方法进行性能比较。

Transformer 实现手写体识别代码如下。

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from einops import rearrange

# 数据准备
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

# Vision Transformer 模型定义
class VisionTransformer(nn.Module):
    def __init__(self, img_size=28, patch_size=7, num_classes=10, dim=128, depth=6, heads=8, mlp_dim=256):
        super(VisionTransformer, self).__init__()
        assert img_size % patch_size == 0, 'Image size must be divisible by the patch size.'

        num_patches = (img_size // patch_size) ** 2
        patch_dim = patch_size * patch_size
        self.patch_size = patch_size

        self.patch_to_embedding = nn.Linear(patch_dim, dim)
        self.pos_embedding = nn.Parameter(torch.randn(1, num_patches + 1, dim))
        self.cls_token = nn.Parameter(torch.randn(1, 1, dim))
        self.transformer = nn.Transformer(dim, nhead=heads, num_encoder_layers=depth)
        self.to_cls_token = nn.Identity()
        self.mlp_head = nn.Sequential(
            nn.LayerNorm(dim),
            nn.Linear(dim, mlp_dim),
            nn.ReLU(),
            nn.Linear(mlp_dim, num_classes)
        )

    def forward(self, x):
        p = self.patch_size
        x = rearrange(x, 'b c (h p1) (w p2) -> b (h w) (p1
        x = self.patch_to_embedding(x)
        b, n, _ = x.shape

        cls_tokens = self.cls_token.expand(b, -1, -1)
        x = torch.cat((cls_tokens, x), dim=1)
        x += self.pos_embedding[:, :(n + 1)]
        x = self.transformer(x, x)
        x = self.to_cls_token(x[:, 0])
        return self.mlp_head(x)
```


图 18a：实现代码

```python
# 模型实例化
model = VisionTransformer()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# 模型训练
for epoch in range(10):  # 训练10个epoch
    model.train()
    for images, labels in train_loader:
        optimizer.zero_grad()
        output = model(images)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()

    print(f'Epoch {epoch + 1}, Loss: {loss.item()}')

# 模型评估
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for images, labels in test_loader:
        output = model(images)
        _, predicted = torch.max(output.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy: {100 * correct / total}%')
```

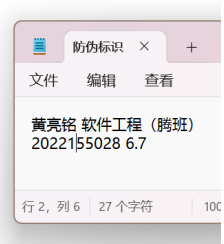

图 18b：实现代码

模型评估结果如下图。

	10	100	1000	5000	10000	15000	20000
人脸识别率	0.125	0.249	0.672	0.817	0.904	0.967	0.993
lossrate	2.324	1.861	0.937	0.489	0.208	0.15	0.128

图 19：评估结果

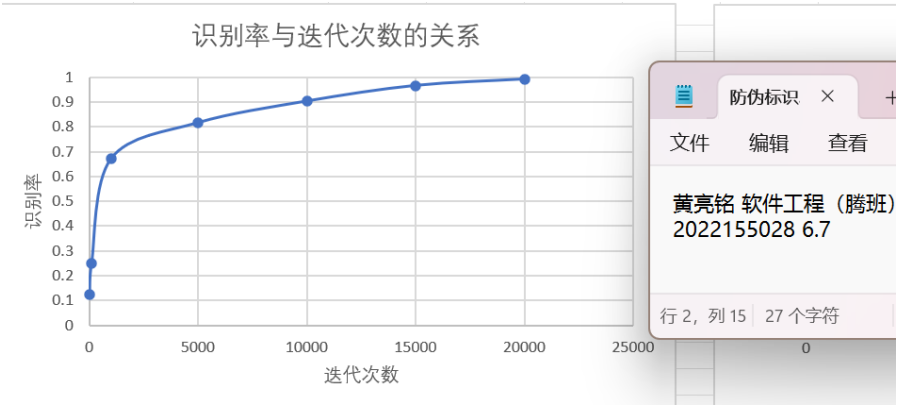

图 20a：可视化评估结果

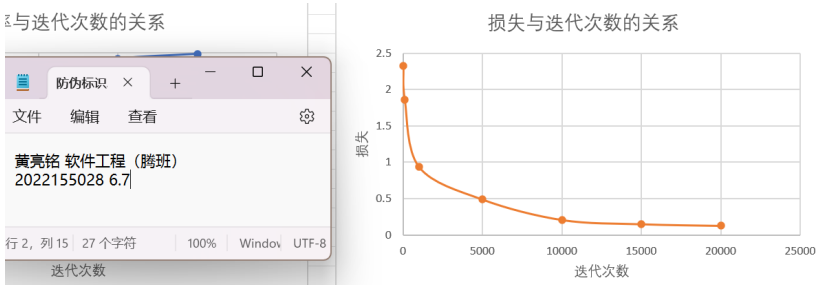

图 20b：可视化评估结果

BP 算法实现手写体识别和 Transformer 实现手写体识别性能比较。

由下图可以发现，当迭代次数较少时，Transformer 的识别率会更高，性能更好。当迭代次数逐渐上升，两者之间的识别率差距越来越小。当迭代次数到达 15000 次时，Transformer 和 BP 算法之间的识别率几乎没有差距。

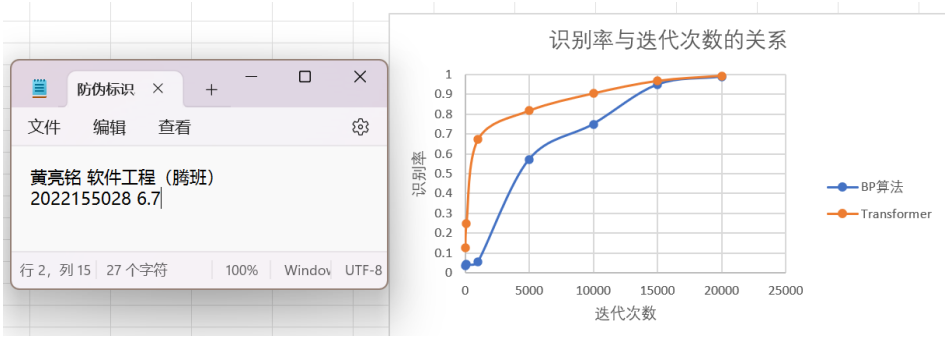

图 21：性能比较可视化

四、实验结论或体会

1. 在本次实验中，我们通过对比 CNN 和全连接神经网络的异同，深入理解了两者在结构设计、连接方式及应用场景上的差异。通过对通用手写体识别任务的实验，我们验证了两者在实际应用中的性能表现。
2. 通过本次实验，我们不仅掌握了 CNN 和 FCN 的理论知识和实现方法，还在实践中体会到了模型选择和优化的重要性。
3. 在实际应用中，选择合适的模型和合理的超参数对实验结果有重要影响。通过不断调整学习率、迭代次数等参数，可以显著提升模型性能。
4. 本次实验不仅巩固了我们对传统神经网络的理解，还让我们接触到 Transformer 等前沿技术，拓展了我们的知识面和应用视野。

指导教师批阅意见： 成绩评定： 指导教师签字： 年　　月　　日
备注：

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。