華中科技大學

研究生课程报告

姓　　名	赖家晨
学　　号	M202474131
系、年级	计算机科技与技术系 2024 级
类　　别	课程报告
报告科目	人工智能

2025 年 1 月 3 日

1　基于产生式的动物识别

1.1　算法原理

产生式系统是一种描述形式化语言的语法，描述若干个不同，但以一个“基本概念”为基础的系统。这个基本概念就是“产生式规则”或产生式的“条件 → 操作”对。如图 1.1所示，一个产生式系统包含事实库、规则集和规则解释 (控制器) 三部分。

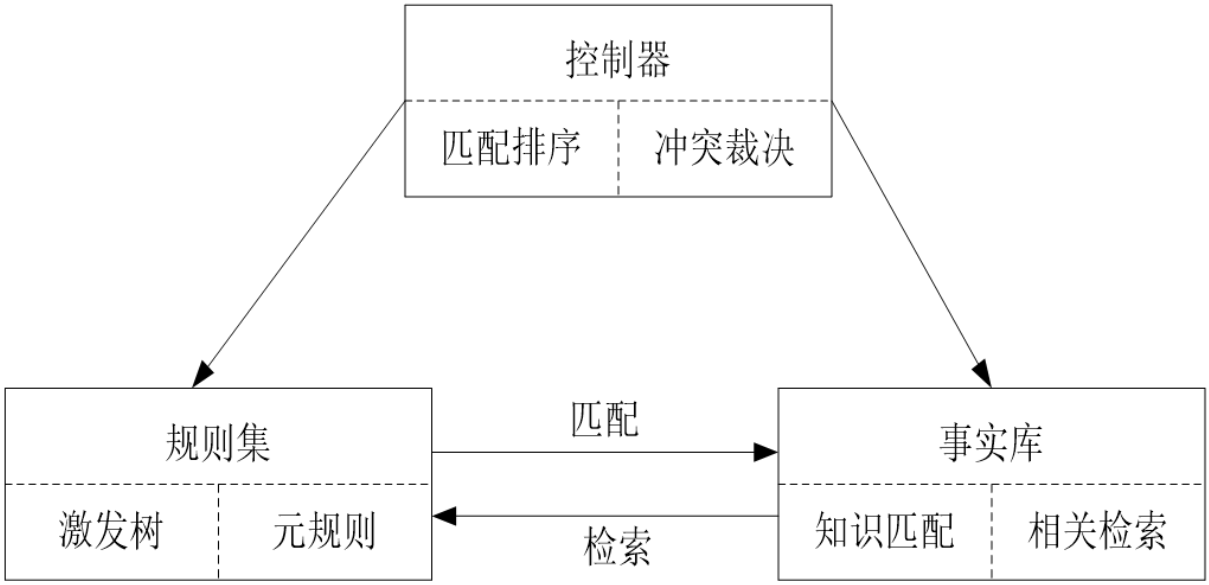

图 1.1 产生式系统

事实库存储已知或可获取的知识，包括推理过程形成的中间结论知识。事实库中的数据,可以是常量、多元数组、谓词或表结构,由规则解释器激活相应的规则。规则库建立并存储问题的状态转移、性质变化等规则,其一般形式为 $P \to Q$ 或 $IF\ P\quad THEN\ Q\quad CF = [0,1]$，其中 P 是产生式的前提或条件，Q 式结论或操作，CF 是确定性因子（置信度)。规则解析（控制器）根据建立的规则，选择相应的控制策略，或是通过规则与事实的匹配，实现知识推理。

产生式系统求解问题的一般步骤如下:

1. 初始化综合数据库，将初始已知事实送入综合数据库 (事实库)。

2. 若规则库 (集) 中有未使用过的规则，且其前提可与综合数据库中的已知事实匹配，继续。若不存在这样的事实，转步骤 5，提供更多事实。

3. 执行当前选中的规则，对该规则做标记，并将执行后得到的结论 (中间结论) 加入综合数据库。若该规则的结论是某些操作，则执行操作。

4. 检查综合数据库中是否已出现问题的解。若出现，求解过程终止；否则转 。

5. 要求提供更多事实，若能提供，则转步骤 2；否则终止。

6. 若规则集中已没有未使用过的规则，则求解过程终止。

1.2 算法步骤

在基于产生式的动物识别中，需要建立一组规则，根据输入的动物的特征信息，推断出动物的种类。因此，首先需要收集识别动物的典型特征信息，包括动物的体型、外观、行为习性等。

在动物识别中，首先，根据识别动物的特征，人为总结一些规则，形成规则集。如：

马：四肢健壮、有蹄、长尾巴、通常用于骑乘或拉车。

鸡：有羽毛、有喙、会下蛋、通常用于提供肉和蛋。

猫：体型较小、有柔软光滑的皮肤、有锋利的牙齿和爪子、是捕鼠高手。

狗：体型多样、有毛发、有锋利的牙齿、是人类忠实的伴侣和守护者。

其次，基于搜集到的动物特征信息，建立一组产生式规则。这些规则可以用于根据输入的动物特征信息，推断出动物的种类。如：

R1：IF 动物有蹄 AND 有长尾巴 AND 用于骑乘或拉车 THEN 动物是马

R2：IF 动物有羽毛 AND 有喙 AND 会下蛋 THEN 动物是鸡

R3：IF 动物体型较小 AND 有柔软光滑的皮肤 AND 有锋利的牙齿和爪子 THEN 动物是猫

R4：IF 动物体型多样 AND 有毛发 AND 有锋利的牙齿 AND 是人类忠实的伴侣和守护者 THEN 动物是狗

最后，实现推理机功能，根据输入的动物特征信息，匹配规则集中规则，并推断出动物的种类。具体的推理机需要实现如下功能：1. 接收输入的动物特征信息。

2. 将输入的动物特征信息，与规则集中的规则进行匹配。

3. 如果找到匹配的规则，则根据规则的结论推断出动物的种类。

4. 如果没有找到匹配的规则，则输出无法识别的提示信息。

1.3 代码实现

首先，我们根据识别动物的特征，人为总结出这些动物的典型特征，并总结得到规则如下：

R1：IF 该动物有毛发 THEN 该动物是哺乳动物

R2：IF 该动物有奶 THEN 该动物是哺乳动物

R3：IF 该动物有羽毛 THEN 该动物是鸟

R4：IF 该动物会飞 AND 会下蛋 THEN 该动物是鸟

R5：IF 该动物吃肉 THEN 该动物是食肉动物

R6：IF 该动物有犬齿 AND 有爪 AND 眼盯前方 THEN 该动物是食肉动物

R7：IF 该动物是哺乳动物 AND 有蹄 THEN 该动物是有蹄类动物

R8：IF 该动物是哺乳动物 AND 是反刍动物 THEN 该动物是有蹄类动物

R9：IF 该动物是哺乳动物 AND 是食肉动物 AND 是黄褐色 AND 身上有暗斑点 THEN 该动物是金钱豹

R10：IF 该动物是哺乳动物 AND 是食肉动物 AND 是黄褐色 AND 身上有黑色条纹 THEN 该动物是虎

R11：IF 该动物是有蹄类动物 AND 有长脖子 AND 有长腿 AND 身上有暗斑点 THEN 该动物是长颈鹿

R12：IF 该动物有蹄类动物 AND 身上有黑色条纹 THEN 该动物是斑马

R13：IF 该动物是鸟 AND 有长脖子 AND 有长腿 AND 不会飞 AND 有黑白二色 THEN 该动物是鸵鸟

R14：IF 该动物是鸟 AND 会游泳 AND 不会飞 AND 有黑白二色 THEN 该动物是企鹅

R15：IF 该动物是鸟 AND 善飞 THEN 该动物是信天翁

其次，我们根据上述规则，实现了推理机。用户可通过界面输入动物的特征信息，系统即可根据输入信息识别出动物的种类，并将结果显示给用户。

具体代码实现如下：

```
from collections import OrderedDict

# 定义规则集
RULES = [
    {"if": {" 毛发": True}, "then": " 哺乳动物"},
    {"if": {" 奶": True}, "then": " 哺乳动物"},
    {"if": {" 羽毛": True}, "then": " 鸟"},
    {"if": {" 飞": True, " 下蛋": True}, "then": " 鸟"},
    {"if": {" 吃肉": True}, "then": " 食肉动物"},
    {"if": {" 锋利牙齿": True, " 锋利爪子": True, " 眼盯前方": True},
    ↪  "then": " 食肉动物"},
    {"if": {" 哺乳动物": True, " 有蹄": True}, "then": " 有蹄类动物"},
    {"if": {" 哺乳动物": True, " 反刍动物": True}, "then": " 有蹄类动物"},
    {"if": {" 哺乳动物": True, " 食肉动物": True, " 黄褐色": True, " 暗斑
    ↪  点":True}, "then": " 金钱豹"},
    {"if": {" 哺乳动物": True, " 食肉动物": True, " 黄褐色": True, " 黑色
    ↪  条纹": True}, "then": " 虎"},
    {"if": {" 有蹄类动物": True, " 长脖子": True, " 长腿": True, " 暗斑
    ↪  点": True}, "then": " 长颈鹿"},
    {"if": {" 有蹄类动物": True, " 黑色条纹": True}, "then": " 斑马"},
    {"if": {" 鸟": True, " 长脖子": True, " 长腿": True, " 不会飞": True,
    ↪  " 黑白二色": True}, "then": " 鸵鸟"},
    {"if": {" 鸟": True, " 游泳": True, " 长腿": True, " 不会飞": True, "
    ↪  黑白二色": True}, "then": " 企鹅"},
```

```
    {"if": {" 鸟": True, " 善飞": True}, "then": " 信天翁"},
]
FeaturesList = []
AnimalFeatures = {}

# 输出全部规则
def printRule():
    print(" 在以下特征中，选取动物特征（输入特征前面的序号，每行输入一个特
    ↪ 征，空行表示输入结束）: ")
    features = []
    for rule in RULES:
        for key, _ in rule["if"].items():
            features.append(key)

    global FeaturesList
    FeaturesList = list(OrderedDict.fromkeys(features))
    for i, key in enumerate(FeaturesList, 1):
        print(f"{i}: {key}", end = " ")
        if i != 0 and i % 8 == 0:
            print()

# 获取输入特征
def getFeature():
    global FeaturesList
    global AnimalFeatures

    for key in FeaturesList:
        AnimalFeatures[key] = False

    line = input()
    while line:
        AnimalFeatures[FeaturesList[int(line) - 1]] = True
        line = input()

# 推理机函数
def inferAnimal():
    for rule in RULES:
```

```
        if all(rule["if"].get(key)  == value for key, value in
        ↪  AnimalFeatures.items() if value == True):
            return rule["then"]
    return " 未知动物"

# 输出全部特征
printRule()

# 获取输入特征
getFeature()

# 识别动物
animal = inferAnimal()
print(f" 识别出的动物是: {animal}")
```

1.4 实验结果

代码运行结果 1 如图 1.2所示，输入为 20：鸟、18：长脖子、19：长腿、21：不会飞和 22：黑白二色，根据给定规则，推测出动物为鸵鸟。

```
在以下特征中，选取动物特征（输入特征前面的序号，每行输入一个特征，空行表示输入结束）：
1: 毛发 2: 奶 3: 羽毛 4: 飞 5: 下蛋 6: 吃肉 7: 锋利牙齿 8: 锋利爪子
9: 眼盯前方 10: 哺乳动物 11: 有蹄 12: 反刍动物 13: 食肉动物 14: 黄褐色 15: 暗斑点 16: 黑色条纹
17: 有蹄类动物 18: 长脖子 19: 长腿 20: 鸟 21: 不会飞 22: 黑白二色 23: 游泳 24: 善飞
20
18
19
21
22

识别出的动物是：鸵鸟
```

图 1.2 实验结果 1

代码运行结果 2 如图 1.3所示，输入为 16：黑色条纹、14：黄褐色、13：食肉动物和 10：哺乳动物，根据给定的规则，推测出动物为虎。

```
在以下特征中，选取动物特征（输入特征前面的序号，每行输入一个特征，空行表示输入结束）：
1: 毛发 2: 奶 3: 羽毛 4: 飞 5: 下蛋 6: 吃肉 7: 锋利牙齿 8: 锋利爪子
9: 眼盯前方 10: 哺乳动物 11: 有蹄 12: 反刍动物 13: 食肉动物 14: 黄褐色 15: 暗斑点 16: 黑色条纹
17: 有蹄类动物 18: 长脖子 19: 长腿 20: 鸟 21: 不会飞 22: 黑白二色 23: 游泳 24: 善飞
16
14
13
10

识别出的动物是：虎
```

图 1.3 实验结果 2

代码运行结果 3 如图 1.4所示，输入为 17：有蹄类动物、15：暗斑点、21：不会飞，根据给定的规则，没有相应的动物符合条件，输出未知动物。

```
在以下特征中，选取动物特征（输入特征前面的序号，每行输入一个特征，空行表示输入结束）：
1: 毛发 2: 奶 3: 羽毛 4: 飞 5: 下蛋 6: 吃肉 7: 锋利牙齿 8: 锋利爪子
9: 眼盯前方 10: 哺乳动物 11: 有蹄 12: 反刍动物 13: 食肉动物 14: 黄褐色 15: 暗斑点 16: 黑色条纹
17: 有蹄类动物 18: 长脖子 19: 长腿 20: 鸟 21: 不会飞 22: 黑白二色 23: 游泳 24: 善飞
17
15
21

识别出的动物是：未知动物
```

图 1.4 实验结果 3

2 基于 CNN 的动物识别

2.1 算法原理

ResNet 即残差网络，它由何恺明、张祥雨、任少卿和孙剑在 2015 年的论文《Deep Residual Learning for Image Recognition》[1] 中首次提出，并可在多种任务中作为骨干网络进行特征提取。它通过跳跃连接的方式较好地解决了梯度消失问题，使得人们得以训练超过 150 层的深度神经网络。常用的 ResNet 结构包括 ResNet-18、ResNet-34、ResNet-50、ResNet-101 和 ResNet-152，每种结构都有着不同的残差块结构和不同数量的残差块，如图 2.1所示。

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

图 2.1 Resnet 模型结构

每种 ResNet 架构都由 5 个卷积层（conv1、conv2_x、conv4_x、conv5_x）、1 个平均池化层、1 个全连接层和 1 个 softmax 层组成。其中的五个卷积层是骨干网络，用于提取图像特征，后面的层被用于图像分类任务。五种 ResNet 架构使用残差块作为基本块。从 conv2_x 到 conv5_x，每个复合卷积层会使用不同数量的残差块。在 ResNet-50 中，conv2_x、conv3_x、conv4_x 和 conv5_x 使用的残差块的数量分别为 3、4、6 和 3。

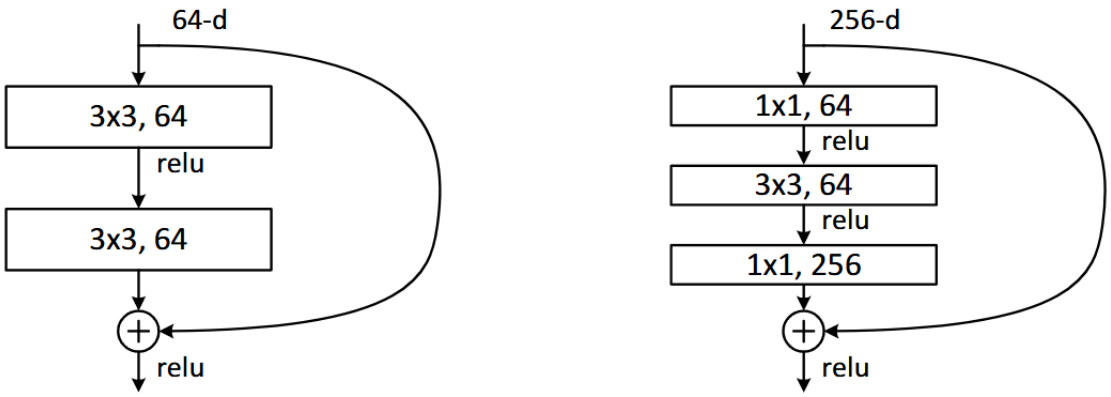

图 2.2 ResNet-50/101/152 的瓶颈结构

此外，从 ResNet-50 开始，网络采用了瓶颈结构，通过交替 1x1 和 3x3 卷积来减少模型参数，并且引入了更多的激活函数，模型的非线性特性和兼容性得到提升。

2.2 代码步骤

1. 超参数设置，这里设置网络训练的超参数包括 epoch、batch size 和学习率。

```
num_epochs = 10
batch_size = 32
learning_rate = 0.001
```

2. 加载数据集，划分训练集和测试集。使用 kaggle 网站上Animals-10 数据集作为模型的数据集，将数据集中的图片预处理后，随机划分成 80% 的训练集和 20% 的测试集。

```
transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],std=[0.229,
    ↪  0.224, 0.225])
])

# 加载数据
dataset =
↪  torchvision.datasets.ImageFolder(root='/kaggle/input/animals10/raw-img',
↪  transform=transform)
```

```
train_size = int(0.8 * len(dataset))
test_size = len(dataset) - train_size
train_set, test_set = torch.utils.data.random_split(dataset,
↪  [train_size, test_size])

train_loader = torch.utils.data.DataLoader(train_set,
↪  batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_set,
↪  batch_size=batch_size, shuffle=False)
```

3. 模型定义，ResNet-50 中有 4 个卷积模块，分别包括了 3 个、4 个、6 个和 3 个残差块。这里直接使用 torchvision 中的 resnet50，并且定义前向传播。

```
class ResNet50(nn.Module):
    def __init__(self, num_classes=10):
        super(ResNet50, self).__init__()
        self.resnet = torchvision.models.resnet50(weights=None)  # 取
        ↪  消预训练权重
        self.fc = nn.Linear(2048, num_classes)  # 修改为 10 个类别

    def forward(self, x):
        x = self.resnet.conv1(x)
        x = self.resnet.bn1(x)
        x = self.resnet.relu(x)
        x = self.resnet.maxpool(x)

        x = self.resnet.layer1(x)
        x = self.resnet.layer2(x)
        x = self.resnet.layer3(x)
        x = self.resnet.layer4(x)

        x = self.resnet.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)

        return x
```

4. 模型参数加载，使用 IMAGENET1K_V1 模型权值初始化模型预训练权重，定义交叉熵为模型的损失函数并且选取 Adam 作为模型的优化器。

```
# 加载预训练模型的 state_dict
state_dict =
↪  torch.load("/kaggle/input/chena-resnet50/resnet50-0676ba61.pth",
↪  map_location=device)

# 创建模型
model = ResNet50(num_classes=10).to(device)

# 只加载除去 fc 层的所有层
pretrained_dict = {k: v for k, v in state_dict.items() if 'fc' not in
↪  k}
model.load_state_dict(pretrained_dict, strict=False)

# 修改 fc 层，防止在加载时出现 fc 层的 shape 不匹配问题
model.fc = nn.Linear(2048, 10)  # 修改为 10 个类别
model = model.to(device)  # 确保模型在设备上

# 定义损失函数和激活函数
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

5. 模型训练，将训练集上的数据移动到 GPU 上，模型进行预测，计算损失函数值，再反向传播，更新模型权重。

```
# 模型训练
total_step = len(train_loader)
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = images.to(device)
        labels = labels.to(device)

        outputs = model(images)
        loss = criterion(outputs, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (i+1) % 100 == 0:
```

```
            print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                   .format(epoch+1, num_epochs, i+1, total_step,
                   ↪  loss.item()))
```

6. 模型评估，将训练得到模型在测试集上进行测试，计算模型的准确度。

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

    print('Test Accuracy of the model on the {} test images: {}
    ↪  %'.format(total, 100 * correct / total))
```

2.3 实验结果

将模型在 kaggle 平台上训练，选取 GPU 为 P100，训练 epoch 为 10，每个 epoch 中有 655 个 step。图 2.3显示部分模型训练过程。

```
Epoch [4/10], Step [500/655], Loss: 1.2123
Epoch [4/10], Step [600/655], Loss: 0.8699
Epoch [5/10], Step [100/655], Loss: 0.8876
Epoch [5/10], Step [200/655], Loss: 1.1089
Epoch [5/10], Step [300/655], Loss: 0.9293
Epoch [5/10], Step [400/655], Loss: 1.1602
Epoch [5/10], Step [500/655], Loss: 0.9036
Epoch [5/10], Step [600/655], Loss: 0.7065
Epoch [6/10], Step [100/655], Loss: 0.7191
Epoch [6/10], Step [200/655], Loss: 1.0287
Epoch [6/10], Step [300/655], Loss: 0.5377
Epoch [6/10], Step [400/655], Loss: 0.6523
Epoch [6/10], Step [500/655], Loss: 1.0495
Epoch [6/10], Step [600/655], Loss: 0.8876
Epoch [7/10], Step [100/655], Loss: 0.6038
Epoch [7/10], Step [200/655], Loss: 0.5140
Epoch [7/10], Step [300/655], Loss: 0.8245
Epoch [7/10], Step [400/655], Loss: 0.6080
Epoch [7/10], Step [500/655], Loss: 0.5028
Epoch [7/10], Step [600/655], Loss: 0.7790
Epoch [8/10], Step [100/655], Loss: 0.5198
Epoch [8/10], Step [200/655], Loss: 0.5820
Epoch [8/10], Step [300/655], Loss: 1.0758
Epoch [8/10], Step [400/655], Loss: 0.7989
Epoch [8/10], Step [500/655], Loss: 0.9637
Epoch [8/10], Step [600/655], Loss: 0.5296
Epoch [9/10], Step [100/655], Loss: 0.6734
Epoch [9/10], Step [200/655], Loss: 0.6327
Epoch [9/10], Step [300/655], Loss: 0.3258
Epoch [9/10], Step [400/655], Loss: 0.3899
Epoch [9/10], Step [500/655], Loss: 0.5101
Epoch [9/10], Step [600/655], Loss: 0.4766
Epoch [10/10], Step [100/655], Loss: 0.7656
Epoch [10/10], Step [200/655], Loss: 0.7179
Epoch [10/10], Step [300/655], Loss: 0.5080
Epoch [10/10], Step [400/655], Loss: 0.4633
Epoch [10/10], Step [500/655], Loss: 0.7348
Epoch [10/10], Step [600/655], Loss: 0.6428
```

图 2.3 训练过程

模型评估如图 2.4所示，模型在 5236 个测试图片上准确度为 76.4%，模型泛化能力不错，训练效果尚可。

在测试集上评估模型的性能

```python
model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print('Test Accuracy of the model on the {} test images: {} %'.format(total, 100 * correct / total))
```

```
Test Accuracy of the model on the 5236 test images: 76.48968678380443 %
```

图 2.4 模型评估

参考文献

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.