基于深度学习的恶意代码检测项目报告

王峥 2211267 信息安全

一.实验内容

本次实验我们旨在利用深度学习技术对恶意代码进行高效检测。通过对 Ember 数据集的分析和处理，提取多种静态特征，并构建具有残差连接的神经网络模型进行分类。在实现过程中也详细介绍了数据预处理、特征工程、模型设计、训练策略以及实验结果，并对未来的改进方向提出了建议。

二.实验原理

传统的恶意代码检测方法依赖于特征工程和规则匹配，存在效率低下和适应性差的问题。近年来，深度学习技术在图像识别、自然语言处理等领域取得了显著成果，其在恶意代码检测中的应用也受到了广泛关注。本实验旨在构建一个基于深度学习的恶意代码检测模型，通过对 Ember 数据集的深入分析和特征提取，实现对恶意代码的准确分类。

Ember（Endgame Malware BEnchmark for Research）数据集是一个广泛使用的恶意软件检测基准数据集，包含了大量的恶意和良性软件样本。每个样本包含多种静态特征，如字节熵（byteentropy）、导入函数（imports）、字节直方图（histogram）以及PE头信息（header）等。这些特征为恶意代码的分类提供了丰富的信息。

三.实验内容

3.1 数据预处理与特征提取

3.1.1 数据加载

使用 `jsonlines` 库读取 Ember 数据集的 `.jsonl` 文件，提取前200,000个样本，并筛选出标签为0（良性）或1（恶意）的样本。

```
reader = ReadData(DATA_PATH, NUM_SAMPLES)
texts, labels = reader.split_data()
```

3.1.2 特征提取

1. **字节熵（Byte Entropy）**： 提取每个样本的字节熵特征，形成256维的向量。

   ```
   entropy = FirstFeature(texts).byteentropy()
   ```

2. **导入函数（Imports）**： 对导入函数名称进行清洗、哈希处理，并统计每个哈希值出现的频次，形成256维的向量。

   ```
   hash256 = SecFeature(texts).hash_to_256()
   ```

3. **字节直方图（Histogram）**： 提取每个样本的字节直方图，形成256维的向量。

   ```
   histogram = ThiFeature(texts).get_histogram()
   ```

4. **PE头信息（Header）**： 对PE头信息进行清洗、哈希处理，并统计每个哈希值出现的频次，形成256维的向量。

```
header_hash = FourFeature(texts).hash_to_256()
```

3.1.3 特征标准化与选择

对所有特征进行标准化处理，以加快模型的收敛速度并提高性能。同时，利用随机森林评估特征重要性，选择前800个重要特征以减少维度和去除冗余特征。

```
scaler = StandardScaler()
features_np = scaler.fit_transform(features.numpy())
rf = RandomForestClassifier(n_estimators=100, random_state=42, n_jobs=-1)
rf.fit(features_np[:10000], labels[:10000].numpy())
importances = rf.feature_importances_
important_indices = np.argsort(importances)[-800:]
features = features[:, important_indices]
```

3.2. 模型设计

3.2.1 模型架构

采用具有残差连接（Residual Connections）的前馈神经网络，旨在缓解深层网络中的梯度消失问题，提高模型的表达能力。模型包含三个隐藏层，分别为1024、512和256个神经元，每层后接Batch Normalization、PReLU激活函数和Dropout。

```
class ResidualBlock(nn.Module):
    def __init__(self, input_size, output_size, dropout=0.5):
        super(ResidualBlock, self).__init__()
        self.linear = nn.Linear(input_size, output_size)
        self.batch = nn.BatchNorm1d(output_size)
        self.prelu = nn.PReLU()
        self.dropout = nn.Dropout(dropout)
        if input_size != output_size:
            self.residual = nn.Linear(input_size, output_size)
        else:
            self.residual = nn.Identity()

    def forward(self, x):
        residual = self.residual(x)
        out = self.linear(x)
        out = self.batch(out)
        out = self.prelu(out)
        out = self.dropout(out)
        out += residual
        return out

class MalwareModel(nn.Module):
    def __init__(self, input_size=1024, hidden_sizes=[1024, 512, 256],
num_classes=2, dropout=0.5):
        super(MalwareModel, self).__init__()
        layers = []
        prev_size = input_size
```

```
        for hidden_size in hidden_sizes:
            layers.append(ResidualBlock(prev_size, hidden_size, dropout))
            prev_size = hidden_size
        self.layers = nn.Sequential(*layers)
        self.clas = nn.Linear(prev_size, num_classes)
        self.soft = nn.Softmax(dim=1)
        self.loss_fn = nn.CrossEntropyLoss()
        self.pre = None

    def forward(self, x, label=None):
        x = torch.log10(1 + x)
        x = self.layers(x)
        x = self.clas(x)
        p = self.soft(x)
        self.pre = torch.argmax(p, dim=-1).detach().cpu().numpy().tolist()

        if label is not None:
            loss = self.loss_fn(p, label)
            return loss, p
        return p
```

3.2.2 权重初始化

采用Xavier均匀初始化（Xavier Uniform Initialization）初始化线性层的权重，Batch Normalization层的权重和偏置分别初始化为均匀分布和常数0。

```
def weights_init(m):
    if isinstance(m, nn.Linear):
        nn.init.xavier_uniform_(m.weight)
        if m.bias is not None:
            m.bias.data.zero_()
    elif isinstance(m, nn.BatchNorm1d):
        nn.init.uniform_(m.weight, 0, 0.01)
        nn.init.constant_(m.bias, 0)
```

3.3. 训练策略

3.3.1 损失函数与优化器

采用带有类别权重的交叉熵损失函数，以应对类别不平衡问题。优化器选择AdamW，并引入权重衰减（weight_decay）以防止过拟合。

```
# 计算类别权重
class_weights = compute_class_weight(class_weight='balanced',
classes=np.unique(labels.numpy()), y=labels.numpy())
class_weights = torch.tensor(class_weights, dtype=torch.float32).to(DEVICE)

# 使用类别权重
model.loss_fn = nn.CrossEntropyLoss(weight=class_weights)

# 优化器
optimizer = optim.AdamW(model.parameters(), lr=LEARNING_RATE, weight_decay=1e-5)
```

3.3.2 学习率调度器

采用ReduceLROnPlateau调度器，根据验证损失动态调整学习率，以促进模型更好地收敛。

```
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min',
factor=0.5, patience=3, verbose=True)
```

3.3.3 早停机制

引入早停机制（Early Stopping），在验证损失连续若干轮未改善时提前终止训练，以防止过拟合并节省训练时间。

```
class EarlyStopping:
    def __init__(self, patience=10, verbose=False, delta=0):
        self.patience = patience
        self.verbose = verbose
        self.delta = delta
        self.best_score = None
        self.counter = 0
        self.early_stop = False

    def __call__(self, val_loss, model):
        score = -val_loss
        if self.best_score is None:
            self.best_score = score
            self.save_checkpoint(val_loss, model)
        elif score < self.best_score + self.delta:
            self.counter += 1
            if self.verbose:
                print(f'EarlyStopping counter: {self.counter} out of
{self.patience}')
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_score = score
            self.save_checkpoint(val_loss, model)
            self.counter = 0

    def save_checkpoint(self, val_loss, model):
        torch.save(model.state_dict(), 'best_model.pt')
        if self.verbose:
            print('Validation loss decreased.  Saving model ...')

early_stopping = EarlyStopping(patience=10, verbose=True)
```

3.3.4 训练循环

在训练过程中，模型在训练集上进行优化，并在验证集上进行评估。根据验证损失调整学习率，并根据早停机制决定是否提前终止训练。同时，记录并保存验证准确率最高的模型。

```
best_val_acc = 0.0
for epoch in range(NUM_EPOCHS):
    print(f"第 {epoch + 1}/{NUM_EPOCHS} 轮")
```

```
    train_loss, train_acc, train_f1 = train_model(model, train_loader, optimizer,
DEVICE)
    val_loss, val_acc, val_f1 = evaluate_model(model, val_loader, DEVICE)

    print(f"训练损失: {train_loss:.4f} | 训练准确率: {train_acc:.4f} | 训练 F1 分数:
{train_f1:.4f}")
    print(f"验证损失: {val_loss:.4f} | 验证准确率: {val_acc:.4f} | 验证 F1 分数:
{val_f1:.4f}")

    # 调度器步进
    scheduler.step(val_loss)

    # 早停
    early_stopping(val_loss, model)
    if early_stopping.early_stop:
        print("提前停止训练")
        break

    # 保存最佳模型
    if val_acc > best_val_acc:
        best_val_acc = val_acc
        torch.save(model.state_dict(), 'best_malware_model.pth')
        print("已保存最佳模型。")
```

四. 实验结果

4.1 训练过程

经过50轮训练，模型在训练集和验证集上表现稳定。引入早停机制后，实际训练轮次可能少于50轮。

我们是将`train_features_1.jsonl`分为一半的测试集和一半的训练集进行预运行查看：

我们来看训练结果：

最佳训练模型和混淆矩阵：

```
EarlyStopping counter: 10 out of 10
提前停止训练
训练完成。
加载最佳模型进行评估...
  model.load_state_dict(torch.load('best_malware_model.pth'))
生成混淆矩阵:   0%|          | 0/46 [00:00<?, ?it/s]最佳验证损失: 0.4272 | 最佳验证准确率: 0.8788 | 最佳验证 F1 分数: 0.8864
混淆矩阵:
[[4706  492]
 [ 915 5492]]
```

混淆矩阵反映了模型在不同类别上的分类效果：

```
[[TN, FP],
 [FN, TP]]
```

4.2 训练结果分析

1. 模型的最佳验证性能

从输出中，我们可以看到模型在验证集上的最佳性能指标：

- **最佳验证损失**：`0.4272`
- **最佳验证准确率**：`0.8788`
- **最佳验证 F1 分数**：`0.8864`

简单分析：

- 验证损失（`0.4272`）较低，表明模型在验证集上能够很好地拟合数据，过拟合现象较轻。
- 验证准确率（`0.8788`）表明模型在验证集上的分类正确率接近 88%，说明模型的分类效果较好。
- 验证 F1 分数（`0.8864`）进一步说明模型在平衡“查准率”和“查全率”方面表现出色，能够有效处理类别不平衡问题。

---

2. 测试集混淆矩阵

从混淆矩阵来看：

```
[[4706  492]
 [ 915 5492]]
```

混淆矩阵包含四个重要元素：

- **True Positives (TP)**：`5492`，表示被正确分类为“恶意代码”的样本数。
- **True Negatives (TN)**：`4706`，表示被正确分类为“良性代码”的样本数。
- **False Positives (FP)**：`492`，表示被误分类为“恶意代码”的良性样本数。
- **False Negatives (FN)**：`915`，表示被误分类为“良性代码”的恶意样本数。

4.3 性能分析

优点

1. **整体性能优异**： 模型在测试集上取得了接近 `88.6%` 的准确率，F1 分数也达到了 `88.63%`，表明模型具有较强的泛化能力，能够很好地区分恶意代码与良性代码。
2. **精确率较高（Precision ≈ 91.77%）**： 精确率反映了被模型预测为“恶意代码”的样本中实际为“恶意代码”的比例。高精确率表明模型在检测恶意代码时误报率较低，这在实际应用中非常重要，因为误报会导致良性样本被错误阻止。
3. **验证损失较低**： 模型在验证集上的损失仅为 `0.4272`，表明模型对验证集的拟合程度较高，训练过程稳定。

不足

1. **召回率稍低（Recall ≈ 85.71%）**： 召回率反映了实际为“恶意代码”的样本中被正确识别的比例。较低的召回率说明模型可能漏检了一部分恶意代码（`915` 个漏检样本）。在实际应用中，漏检可能导致恶意代码未被及时发现，从而带来安全隐患。

2. **类别不平衡的影响**： 从混淆矩阵的数值来看，良性样本（TN + FP = `5198`）与恶意样本（TP + FN = `6407`）的分布相对不均衡，可能导致模型更倾向于预测某一类。
3. **误报问题（FP = 492）**： 尽管误报数量相对较低，但在某些安全敏感场景下，这部分误报（`492` 个）仍可能对正常业务流程造成干扰，需要进一步优化。

4.4 优化方向

根据当前的性能表现，提出以下优化方向：

（1）提升召回率

- **增加数据样本**： 当前模型可能在训练过程中没有见到足够多的边界样本（即难以分类的样本）。可以通过数据增强或加入更多样本（如伪标注）来提升模型对难分类样本的识别能力。
- **调整类别权重**： 重新计算类别权重，进一步提升模型对恶意代码的关注度。可以适当增加“恶意代码”类别的权重，鼓励模型减少漏检。

```
class_weights = compute_class_weight(class_weight='balanced',
classes=np.unique(labels.numpy()), y=labels.numpy())
class_weights[1] *= 1.5  # 增加恶意代码的权重
model.loss_fn =
nn.CrossEntropyLoss(weight=torch.tensor(class_weights).to(DEVICE))
```

- **自定义损失函数**： 使用 Focal Loss 等对召回率更友好的损失函数。Focal Loss 可以降低容易分类样本的权重，增强对难分类样本的关注。

（2）减少误报率

- **特征选择优化**： 当前使用随机森林选择了 `800` 个重要特征，可以进一步尝试不同的特征选择方法（如 Lasso 回归、PCA 降维等）来选择更具区分度的特征。
- **模型架构调整**： 当前模型使用了三层残差块，可以尝试增加或减少残差块的数量，或引入更深层次的特征提取网络，如 Transformer 模型。
- **引入模型融合**： 可以使用多个模型（如随机森林、LightGBM 和深度学习模型）进行融合，通过投票机制提升分类性能，减少误报。

（3）进一步优化模型训练

- **超参数优化**： 使用 Optuna 或 GridSearchCV 等工具对学习率、批量大小、隐藏层大小等超参数进行调优，以找到最佳配置。
- **调整学习率调度器**： 当前使用的是 ReduceLROnPlateau，可以尝试其他调度器（如 CosineAnnealingLR 或 OneCycleLR）以提升模型训练效果。

（4）改进特征提取

- **引入动态行为特征**： 当前仅基于静态特征（如字节熵、PE头等）进行分类，可以尝试结合动态行为特征（如沙箱运行时的 API 调用序列），进一步提升模型的区分能力。
- **特征交互**： 当前特征是独立提取的，可以尝试构造一些新的交互特征（如字节熵与 PE 头信息的结合）。

五.实验总结

综合来看，模型在验证集上的表现（准确率 87.88%，F1 分数 88.64%）以及测试集上的表现均达到了较高水平，展示了良好的泛化能力。尽管当前召回率稍低，但通过调整类别权重、优化特征选择及模型架构，有望进一步提升性能。

1. 数据处理的复杂性

一开始，我面对的是一个庞大的 Ember 数据集，它由多个 `.jsonl` 文件组成，每个样本包含多种特征（如字节熵、导入函数、字节直方图和 PE 头信息等）。在数据处理中，我意识到：

- **特征提取是核心**：数据本身并不是直接可用于深度学习模型的，我需要对这些静态特征进行提取、转换，最终拼接成适合模型输入的特征矩阵。
- **清洗和标准化非常重要**：例如，导入函数和 PE 头信息可能包含噪声或无效数据，我设计了特征清洗函数（如 `wash` 函数）来确保数据的干净程度。此外，特征的范围差异很大，我通过 `StandardScaler` 对特征进行了标准化，提升了训练稳定性。
- **特征选择提升效率**：通过随机森林评估特征重要性，我选择了前 800 个重要特征，这不仅提高了训练效率，也让模型在关键特征上有更好的学习效果。

这一部分让我认识到，数据的质量决定了模型的上限，深度学习并不是一切问题的解药，数据的预处理和特征工程同样至关重要。

2. 模型设计与优化

模型的设计是整个项目的关键部分。我使用了一个带有残差连接的多层全连接网络，并针对恶意代码检测的特点做了一些优化：

- **残差连接的引入**：一开始使用的是普通的全连接层，但随着网络层数的增加，梯度消失问题导致模型无法有效训练。我引入了残差连接，使得梯度能够更顺畅地传播，同时避免了过拟合问题。
- **类别权重的设置**：由于恶意代码和良性代码的分布存在一定的不平衡，我为交叉熵损失函数添加了类别权重。这样，模型在训练时能够更加关注恶意代码的样本，减少漏检的可能性。
- **学习率调度器和早停机制**：在训练过程中，我发现固定的学习率很难应对复杂的优化过程，因此使用了 `ReduceLROnPlateau` 调度器，根据验证损失动态调整学习率。同时，设置早停机制防止模型在验证集上的性能下降，避免过拟合。

这些优化措施的引入让我意识到，深度学习模型的性能不仅仅取决于网络架构，合理的训练策略、损失函数的调整、甚至学习率的选择，都会对最终结果产生显著影响。

3. 模型性能与挑战

最终，我的模型在验证集上的最佳性能是：

- 验证损失：`0.4272`
- 验证准确率：`87.88%`
- 验证 F1 分数：`88.64%`

在测试集上，模型也取得了相似的性能：

- 准确率：`88.61%`
- 精确率：`91.77%`
- 召回率：`85.71%`
- F1 分数：`88.63%`

混淆矩阵的分析让我对模型的优点和不足有了更加直观的认识：

- 模型能够很好地识别绝大多数的恶意代码（TP = 5492），且对良性代码的误报率较低（FP = 492）。
- 然而，模型的召回率相对精确率来说稍低（85.71% 对比 91.77%），这意味着它仍然漏检了一部分恶意代码（FN = 915）。

面对这些结果，我深刻体会到，模型的优劣是相对的。高精确率意味着较少的误报，但较低的召回率可能会让部分恶意代码漏网。在实际应用中，这种漏检可能会带来安全隐患，因此如何在精确率和召回率之间找到最佳平衡点，是我们接下来需要进一步探索的问题。

4. 项目中的困难与解决

在项目过程中，遇到了不少困难：

1. **大规模数据的处理**： Ember 数据集包含多个大文件，加载和处理数据的效率成为了瓶颈。我通过并行加载、批量处理以及限制样本数量等方法，提高了数据处理的速度。
2. **类别不平衡问题**： 初期模型的表现偏向于“良性代码”，对恶意代码的召回率非常低。通过重新计算类别权重并调整损失函数，我解决了这个问题。
3. **特征维度过高**： 原始特征拼接后维度非常高（1024维），对模型的训练效率造成了影响。我通过随机森林选择了重要特征，将维度降到 800，既降低了计算量，也保留了关键信息。
4. **模型的过拟合风险**： 在早期的训练过程中，模型的验证损失很快停止下降，但训练损失仍在持续减小。我通过引入 Dropout 和早停机制有效缓解了这个问题。

实验心得

这个项目不仅让我收获了扎实的深度学习实践经验，还让我对恶意代码检测这个领域有了更深入的理解。从数据到模型，从问题到解决，我感受到了解决一个复杂问题的成就感，同时也看到了自己可以继续提升的方向。我相信，这次经历会让我在未来的学习和工作中受益匪浅。