**基于Visual Studio Tools for AI的TensorFlow编程实现**

**基于监督学习的恶意攻击特征识别**

目 录

[1. 背景介绍 3](#_Toc520148557)

[1.1 恶意攻击 3](#_Toc520148558)

[1.2 Tensorflow 3](#_Toc520148559)

[2. 相关技术简介 4](#_Toc520148560)

[2.1 深度神经网络简介 4](#_Toc520148561)

[2.2 Keras简介 4](#_Toc520148562)

[2.3 Eager Execution简介 4](#_Toc520148563)

[2.4 数据集KDD cup 1999介绍 4](#_Toc520148564)

[3. 实现目标 6](#_Toc520148565)

[4. 实现步骤 7](#_Toc520148566)

[4.1 数据集预处理 7](#_Toc520148567)

[4.1.1 细粒度识别数据集预处理 7](#_Toc520148568)

[4.1.2 粗粒度识别数据集预处理 8](#_Toc520148569)

[4.2 基于Keras库的Sequential模型和Eager Execution动态图机制实现细粒度识别 9](#_Toc520148570)

[4.2.1 载入所需要的库进行预操作 9](#_Toc520148571)

[4.2.2 构建神经网络 10](#_Toc520148572)

[4.2.3 进行训练，构建模型 10](#_Toc520148573)

[4.2.4 使用测试集带入所建立模型，并计算预测准确度 11](#_Toc520148574)

[4.3 基于DNN实现粗粒度识别 12](#_Toc520148575)

[4.3.1 将训练集导入容器 12](#_Toc520148576)

[4.3.2 建立DNN网络 12](#_Toc520148577)

[4.3.3 进行训练建立模型 12](#_Toc520148578)

[4.3.4 根据训练好的模型对测试集进行预测，根据结果改进模型 13](#_Toc520148579)

[5. 实验结果 14](#_Toc520148580)

[5.1 细粒度识别结果 14](#_Toc520148581)

[5.2 粗粒度识别结果 14](#_Toc520148582)

[6. 总结 15](#_Toc520148583)

[7. 附录 16](#_Toc520148584)

[7.1 KDD cup 1999 数据特征描述 16](#_Toc520148585)

[7.2 细粒度识别运行结果截图 17](#_Toc520148586)

[7.3 粗粒度识别运行结果截图 18](#_Toc520148587)

1. 背景介绍

## 1.1 恶意攻击

恶意攻击是指在[信息安全技术](https://baike.baidu.com/item/%E4%BF%A1%E6%81%AF%E5%AE%89%E5%85%A8%E6%8A%80%E6%9C%AF/2253817" \t "_blank)中，内部人员有计划地窃听、偷窃、或损坏信息，或拒绝其他授权用户的访问。恶意攻击自从产生到现在，已经危害了无数的电脑网络，给人们的造成了严重的经济损失。对恶意攻击的识别作为一项可以极大降低恶意攻击危害的手段，一直受到高度关注。在本项目中，将会使用较为新颖的机器学习方法，对恶意攻击进行识别，为对恶意攻击的研究、预防提供基础。

在本项目中，我们使用的数据中攻击主要有4大类共39种攻击，四种攻击类型为：

1. DOS, denial-of-service.拒绝服务攻击，例如ping-of-death, syn flood, smurf等；
2. R2L, unauthorized access from a remote machine to a local machine. 来自远程主机的未授权访问，例如guessing password；
3. U2R, unauthorized access to local superuser privileges by a local unpivileged user. 未授权的本地超级用户特权访问，例如buffer overflow attacks；
4. PROBING, surveillance and probing, 端口监视或扫描，例如port-scan, ping-sweep等。

具体种类见2.4节数据集介绍。

## File:TensorFlowLogo.svg1.2 Tensorflow

TensorFlow™ 是一个开放源代码软件库，用于进行高性能数值计算。借助其灵活的架构，用户可以轻松地将计算工作部署到多种平台（CPU、GPU、TPU）和设备（桌面设备、服务器集群、移动设备、边缘设备等）。TensorFlow™ 最初是由 Google Brain 团队（隶属于 Google 的 AI 部门）中的研究人员和工程师开发的，可为机器学习和深度学习提供强力支持，并且其灵活的数值计算核心广泛应用于许多其他科学领域。

1. 相关技术简介

## 2.1 深度神经网络简介

神经网络是一组大致模仿人类大脑构造设计的算法，用于识别模式。神经网络通过机器感知系统解释传感器数据，对原始输入进行标记或聚类。神经网络所能识别的模式是包含在向量中的数值形式，因此图像、声音、文本、时间序列等一切现实世界的数据必须转换为数值。

深度神经网络(Deep Neural Network,DNN)可以理解为除一个输入层与一个输出层以外具有很多隐藏层的神经网络，DNN的大致结构如下图所示:

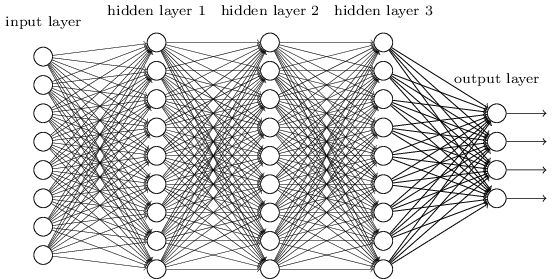


图 2.1 深度神经网络结构图

## 2.2 Keras简介

Keras 是一个用 Python 编写的高级神经网络 API，它可以以Tensorflow作为后端运行，Keras库能够很快的运算出实验结果，对本次实验有较大帮助。

特点：

* 用户友好。 Keras 是为人类而不是为机器设计的 API。它把用户体验放在首要和中心位置。Keras 遵循减少认知困难的最佳实践：它提供一致且简单的 API，将常见用例所需的用户操作数量降至最低，并且在用户错误时提供清晰和可操作的反馈。
* 模块化。 模型被理解为由独立的、完全可配置的模块构成的序列或图。这些模块可以以尽可能少的限制组装在一起。特别是神经网络层、损失函数、优化器、初始化方法、激活函数、正则化方法，它们都是可以结合起来构建新模型的模块。
* 易扩展性。 新的模块是很容易添加的（作为新的类和函数），现有的模块已经提供了充足的示例。由于能够轻松地创建可以提高表现力的新模块，Keras 更加适合高级研究。
* 基于 Python 实现。 Keras 没有特定格式的单独配置文件。模型定义在 Python 代码中，这些代码紧凑，易于调试，并且易于扩展。

## 2.3 Eager Execution简介

TensorFlow 的 Eager Execution 是一种命令式编程环境，可立即评估操作，无需构建图：操作会返回具体的值，而不是构建以后再运行的计算图（动态图模型）。

## 2.4 数据集KDD cup 1999介绍

此数据集首先是美国国防部高级规划署（DARPA）在MIT林肯实验室进行的研究收集数据随后来自哥伦比亚大学的Sal Stolfo 教授和来自北卡罗莱纳州立大学的 Wenke Lee 教授采用数据挖掘等技术对以上的数据集进行特征分析和数据预处理，形成了一个新的数据集。该数据集用于1999年举行的KDD CUP竞赛中，成为著名的KDD99数据集。虽然年代有些久远，但KDD99数据集仍然是网络入侵检测领域的事实Benckmark，为基于计算智能的网络入侵检测研究奠定基础。

训练集大概包含5,000,000多个网络连接记录，测试数据大概包含2,000,000个网络连接记录。每个网络连接被标记为正常（normal）或异常（attack），异常类型被细分为4大类共39种攻击类型，其中22种攻击类型出现在训练集中，另有17种未知攻击类型出现在测试集中。

KDD99数据集中每个连接（\*）用41个特征来描述，加上最后的标记（label），一共有42项：如下所示

2, tcp, smtp, SF, 1684, 363, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0.00, 0.00, 0.00, 0.00, 1.00, 0.00, 0.00, 104, 66, 0.63, 0.03, 0.01, 0.00, 0.00, 0.00, 0.00, 0.00, normal.

0, tcp, private, REJ, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 38, 1, 0.00, 0.00, 1.00, 1.00, 0.03, 0.55, 0.00, 208, 1, 0.00, 0.11, 0.18, 0.00, 0.01, 0.00, 0.42, 1.00, portsweep.

41项特征分为4大类，具体介绍见附录7.1。

1. 实现目标

由于网络恶意行为多种多样且在不断出现新的攻击方式，且有些攻击类型数量十分稀少，如果对每一个攻击类型都进行训练，构建模型需要消耗大量的资源，且准确度不高，且无法保证完全覆盖一些新出现的或者数量较少的攻击类型。我们可以使用如下策略：首先将所有类型的攻击标记恶意行为标签，其他行为标记正常标签，通过相关算法识别恶意行为与正常行为，并可以识别在训练集中没有的，新出现的恶意行为。但是此方法损失了数据集攻击类型的信息，造成了信息的浪费。

由此，本项目使用如下策略：将出现频率最高的几种恶意行为进行训练形成一个模型，此模型可以准确识别出攻击类型，但是仅能识别出特定的几种，我们将其称为细粒度识别，即，可以识别出具体攻击类型的模型。将这几种攻击之外恶意行为统一标记为恶意行为标签，进行训练得出可以分辨恶意行为与正常行为的二分类模型，将此模型称为粗粒度识别即，只能识别出是恶意攻击还是正常行为的模型。需要注意的是此模型不包含细粒度识别中的攻击类型，目的是缩短训练时间，提高准确度。

在识别时首先使用细粒度识别，如果无法识别为细粒度识别中包含的任意一种攻击类型，再使用粗粒度识别。得到的结果有两种情况：1. 具体的攻击类型；2. 恶意行为或正常行为标签。由此即可形成一种基于Tensorflow的有监督恶意行为特征识别方案。

根据我们数据集的实际情况，我们的细粒度识别可以识别的四种攻击类型为：

1. Ipsweep攻击：IPsweep：严格的说，IPsweep并不能算是正式的攻击，IPsweep的过程只是向各个地址发送ping包等待回应，用以探测网络中存活的主机，从而为下一步的攻击做准备。
2. Neptune(SYN flood)攻击：利用了TCP三次握手中的弱点。具体的方法是：攻击主机向目标主机发出SYN报文发起TCP连接，但是攻击主机并不回复最后一个ack报文，使得TCP握手无法完成，而目标主机在TCP握手中需要分配资源维持这些未完成的连接，当这样的连接到达一定数目后，目标主机就无法再响应其他的连接请求。构造出Synflood报文比较容易，只要将syn位置1，然后连接目标主机的某个可用服务的端口即可。
3. Smurf攻击： Smurf攻击通过使用将回复地址设置成受害网络的[广播地址](https://baike.baidu.com/item/%E5%B9%BF%E6%92%AD%E5%9C%B0%E5%9D%80" \t "_blank)的ICMP应答请求(ping)数据包，来淹没受害主机，最终导致该网络的所有主机都对此ICMP应答请求做出答复，导致[网络阻塞](https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E9%98%BB%E5%A1%9E" \t "_blank)。更加复杂的Smurf将源地址改为第三方的受害者，最终导致第三方崩溃。
4. Portsweep攻击：一种端口扫描方式。其方法是对指定目标的端口发送SYN报文发起TCP连接，如果主机返回的是SYN+ACK，那么表示这个端口上有相应的服务开放，如果主机返回的是RST，那么说明这个端口没有服务在监听。

其他类型的攻击统一归入粗粒度识别中，标记为恶意行为。

1. 实现步骤

## 4.1 数据集预处理

### 4.1.1 细粒度识别数据集预处理

我们的数据集使用的是KDD cup 1999，为了测试方案能否识别全新的恶意行为，在测试集中包含训练集中所没有的攻击类型。

本次实验中，我们一共选取了四种最具有代表性同时也是数据集中出现次数最多的4种攻击方式，分别为Ipsweep攻击，Neptune攻击，Smurf攻击，以及Portsweep攻击，加上正常访问的网络行为，一共有5种不同的网络访问行为，将数据集合中带有这四种攻击的访问条目提取出来组成新的数据集，具体实现如下：

1. **import** csv
3. **def** f\_to\_csv():
4. file = open("G:\\TensorflowPJ\\kddcup.data\_10\_percent\_corrected.csv","r")
5. csvfile = open('G:\\TensorflowPJ\\Dataget\\train\_data\_4K.csv','w',newline='')
6. list = []
7. swriter = csv.writer(csvfile,dialect='excel')
8. **for** line **in** file:
9. line\_list = line.strip().split(',')
10. **if** line\_list[41] == 'normal.':
11. line\_list[41] = '0'
12. **for** i **in** range(42):
13. list.append(line\_list[i])
14. **elif** line\_list[41] == 'ipsweep.':
15. line\_list[41] = '1'
16. **for** i **in** range(42):
17. list.append(line\_list[i])
18. **elif** line\_list[41] == 'neptune.':
19. line\_list[41] = '2'
20. **for** i **in** range(42):
21. list.append(line\_list[i])
22. **elif** line\_list[41] == 'smurf.':
23. line\_list[41] = '3'
24. **for** i **in** range(42):
25. list.append(line\_list[i])
26. **elif** line\_list[41] == 'portsweep.':
27. line\_list[41] = '4'
28. **for** i **in** range(42):
29. list.append(line\_list[i])
30. **if** list:
31. swriter.writerow(list)
32. list = []
34. **if** \_\_name\_\_ == '\_\_main\_\_':
35. f\_to\_csv()

在原始KDD CUP 99数据集中，不同网络访问行为的标签以字符串形式进行存储，因此，我们在将数据转换成CSV格式文件时将原始数据中的标签转换为了可供Tensorflow识别并使用的int型数据(0~5)来分别代表不同的网络访问行为。

由于细粒度识别只识别特定的4种攻击，特征较为明显，在测试后发现，每种类型只需要600条，包含正常类别，训练集一共只需要3000条访问条目即可产生识别较为准确的模型。测试集则使用提取的全部数据为28345条访问数据。

### 4.1.2 粗粒度识别数据集预处理

粗粒度数据集将上述4种类型的访问条目剔除。余下的数据集分为恶意行为与正常行为两种，其中恶意行为使用1表示，正常行为使用0表示，处理完成后，训练集大小为494021条访问信息，测试集大小为311029条访问信息。具体代码如下：

1. **def** Transfor\_Data():
2. **print**("Transfor\_Data() Starting")
4. in\_file = open(TRANING\_FILE, 'r', encoding='UTF-8')
5. out\_file = open(DEALED\_TRAINING\_DATA, 'w', newline="")
6. out\_writer = csv.writer(out\_file)
8. #第一行空出留最后写统计数据
9. out\_writer.writerow([])
11. #开始按行读取，提出所需特征值，写入csv
12. dealed\_data = []
13. line = in\_file.readline()
15. **while** line:
16. #4~9行是TCP连接基本特征, 共9种
17. #10~22是TCP连接的内容特征，共13种
18. #23~30是基于时间的网络流量特征统计，共9种
19. #31~41基于主机的网络流量统计特征, 共10种
20. #提取37种特征
21. line = line.split(',')
22. **for** i **in** range(4,41):
23. dealed\_data.append(line[i])
24. **if**(line[41].strip() == "normal."):
25. #如果不是攻击标记为0，是攻击标记为1
26. dealed\_data.append(0)
27. **else**:
28. dealed\_data.append(1)
29. #将dealed\_data写回文件
30. out\_writer.writerow(dealed\_data)
31. dealed\_data = []
32. #下一轮读取
33. line = in\_file.readline()
35. in\_file.close()
36. out\_file.close()
37. **print**("Transfor\_Data() Completed")

## 4.2 基于Keras库的Sequential模型和Eager Execution动态图机制实现细粒度识别

具体实验步骤：

1. 载入所需要的库并进行预操作；
2. 构建神经网络；
3. 进行训练，建立模型；
4. 使用测试集带入所建立模型，并计算预测准确度，同时对模型进行改进。

### 4.2.1 载入所需要的库进行预操作

细粒度识别中训练集一共3000条访问条目，测试集则使用提取的全部数据为28345条访问数据。本次实验所需要的库如下所示：

1. **from** \_\_future\_\_ **import** absolute\_import, division, print\_function
3. **import** os
4. **import** matplotlib.pyplot as plt
6. **import** tensorflow as tf
7. **import** tensorflow.contrib.eager as tfe
9. tf.enable\_eager\_execution() #启动eager execution动态图模块

设置训练集路径

1. train\_dataset\_fp = "G:\\TensorflowPJ\\Keras\_N\\train\_data\_4K\_simple.csv"

由于我们的数据集是 CSV 格式的文本文件，因此我们会将特征和标签值解析为 Python 模型能够使用的格式。在这里我们定义parse\_csv函数，系统会将文件中的每一行传递给prase\_csv函数，该函数会获取前四个特征字段，并将它们合并为一个张量。然后，系统会将最后一个字段解析为标签。该函数会返回特征和标签这两个张量：

1. **def** parse\_csv(line):
2. example\_defaults = [[0.], [0.], [0.], [0.], [0.], [0.], [0.], [0.], [0.], [0.],[0.], [0.],
3. [0.],[0.], [0.], [0.],[0.], [0.], [0.],[0.], [0.], [0.],[0.], [0.], [0.],[0.], [0.],
4. [0.],[0.], [0.], [0.],[0.], [0.], [0.],[0.], [0.], [0.],[0.], [0.],[0.], [0]]  # 设置数据格式
5. parsed\_line = tf.decode\_csv(line, example\_defaults)
6. # 前37列是特征值 构成一个TENSOR 来自于KDD CUP 99数据集
7. features = tf.reshape(parsed\_line[:-1], shape=(38,))
8. #最后一列是数据标签 代表不同的攻击（目前有5种：4种不同攻击和正常访问）
9. label = tf.reshape(parsed\_line[-1], shape=())
10. **return** features, label

在此处我们使用tf.data.TextLineDataset 加载 CSV 格式的文本文件，并且使用prase\_csv解析该文件。tf.data.Dataset 将输入管道表示为一组元素以及作用于这些元素的一系列转换。转换方法会链接在一起或按顺序调用，只需确保保留对返回的 Dataset 对象的引用即可。

1. #创建训练用的dataset
2. train\_dataset = tf.data.TextLineDataset(train\_dataset\_fp)
3. train\_dataset = train\_dataset.skip(1)
4. train\_dataset = train\_dataset.map(parse\_csv)

由于随机排列的样本能够实现最好的训练效果，因此我们使用f.data.Dataset.shuffle 将条目随机化，并将 buffer\_size 设为大于样本数。为了更快速地训练模型，将数据集的批次大小设为一次训练 32 个样本。

1. train\_dataset = train\_dataset.shuffle(buffer\_size=3001)
2. #设置一个缓冲区 将样本进行随机化处理 缓冲区大小要大于数据集总数
3. train\_dataset = train\_dataset.batch(32)  #批处理设置 加快运行速度

### 4.2.2 构建神经网络

接下来，我们使用keras库来创建我们所需要的神经网络模型，此处我们选择Sequential模型。tf.keras.Sequential模型是层的线性层叠。该模型的构造函数会接受一系列层实例；在我们的程序中，接受的是 2 个密集层（分别包含 50 个节点）以及 1 个输出层（包含 5个代表标签预测的节点）。第一个层的 input\_shape 参数对应该数据集中的特征数量，它是一项必需参数。

1. #设置神经网络模型 此处使用了Keras库中的顺序模型Sequential 激活函数为relu 两个隐藏层分别有50个神经元节点
2. model = tf.keras.Sequential([
3. tf.keras.layers.Dense(50, activation="relu", input\_shape=(38,)),
4. tf.keras.layers.Dense(50, activation="relu"),
5. tf.keras.layers.Dense(5)
6. ])

随后，我们定义损失函数和梯度下降函数。在模型的建立阶段和评估阶段，我们都需要计算模型的损失。这样可以衡量模型的预测结果与预期标签有多大偏差，也就是说，模型的效果有多差。我们希望尽可能减小或优化这个值。

1. #定义损失函数
2. **def** loss(model, x, y):
3. y\_ = model(x)
4. **return** tf.losses.sparse\_softmax\_cross\_entropy(labels=y, logits=y\_)
6. #定义梯度下降函数
7. **def** grad(model, inputs, targets):
8. with tf.GradientTape() as tape:
9. loss\_value = loss(model, inputs, targets)
10. **return** tape.gradient(loss\_value, model.variables)

接下来，我们创建一个optimizer来逐渐减少损失值，参数为学习率，即为每次梯度下降度步长，代码如下：

1. optimizer = tf.train.GradientDescentOptimizer(learning\_rate=0.001) #学习率 即为梯度下降的步长

### 4.2.3 进行训练，构建模型

下面开始模型训练，在每个训练周期内，获取每个网络访问样本的特征值和标签，同时根据样本的特征值进行预测，然后比较预测结果和标签，根据比较结果来衡量预测结果的错误率，并利用所得到的值计算模型的损失和下降梯度，同时在每一次循环结束之后输出本次预测结果的损失和精度。代码实现如下：

1. train\_loss\_results = []
2. train\_accuracy\_results = []
4. num\_epochs = 6   #迭代次数
5. #进行迭代训练
6. **for** epoch **in** range(num\_epochs):
7. epoch\_loss\_avg = tfe.metrics.Mean()
8. epoch\_accuracy = tfe.metrics.Accuracy()
10. **for** x, y **in** train\_dataset:
11. grads = grad(model, x, y)
12. optimizer.apply\_gradients(zip(grads, model.variables),
13. global\_step=tf.train.get\_or\_create\_global\_step())
15. epoch\_loss\_avg(loss(model, x, y))
16. epoch\_accuracy(tf.argmax(model(x), axis=1, output\_type=tf.int32), y)
18. train\_loss\_results.append(epoch\_loss\_avg.result())
19. train\_accuracy\_results.append(epoch\_accuracy.result())
21. **if** epoch % 1 == 0:
23. **print**("Epoch {:03d}: Loss: {:.3f}, Accuracy: {:.3%}".format(epoch,
24. epoch\_loss\_avg.result(),
25. epoch\_accuracy.result()))

为了得到一个更直观的结果，我们将每一次循环后的损失和精度画出函数图像，以便我们调整神经网络中神经元的个数以及梯度下降的步长这些参数。代码如下：

1. #做出损失率和成功率随迭代次数变化的图像
2. fig, axes = plt.subplots(2, sharex=True, figsize=(12, 8))
3. fig.suptitle('Training Metrics')
5. axes[0].set\_ylabel("Loss", fontsize=14)
6. axes[0].plot(train\_loss\_results)
8. axes[1].set\_ylabel("Accuracy", fontsize=14)
9. axes[1].set\_xlabel("Epoch", fontsize=14)
10. axes[1].plot(train\_accuracy\_results)
12. plt.show()

### 4.2.4 使用测试集带入所建立模型，并计算预测准确度

下面我们选取另外一个测试集进行测试，并输出测试精度，代码如下：

1. #设置测试集
2. test\_fp = "G:\\TensorflowPJ\\Keras\_N\\test\_data\_4K.csv"

5. test\_dataset = tf.data.TextLineDataset(test\_fp)
6. test\_dataset = test\_dataset.skip(1)
7. test\_dataset = test\_dataset.map(parse\_csv)
8. #test\_dataset = test\_dataset.shuffle(1001)     由于此处的数据已经随机化，为加快运行效率，此处不做处理
9. test\_dataset = test\_dataset.batch(10000)
11. test\_accuracy = tfe.metrics.Accuracy()
12. #进行测试
13. **for** (x, y) **in** test\_dataset:
14. prediction = tf.argmax(model(x), axis=1, output\_type=tf.int32)
15. test\_accuracy(prediction, y)
17. **print**("Test set accuracy: {:.3%}".format(test\_accuracy.result()))

## 4.3 基于DNN实现粗粒度识别

1. 对数据进行预处理，规范数据格式，方便后续步骤处理；
2. 将训练集导入容器；
3. 建立DNN网络；
4. 进行训练建立模型；
5. 根据训练好的模型对测试集进行预测，根据结果改进训练模型。

### 4.3.1 将训练集导入容器

所需数据提取后，使用tf.contrib.learn.datasets.base.load\_csv\_with\_header（）函数读取训练集，放入容器。同时指定数据类型，将数据集内容规范化。

1. training\_set = tf.contrib.learn.datasets.base.load\_csv\_with\_header(
2. filename=DNN\_TRAINING,
3. target\_dtype=np.int,
4. features\_dtype=np.float32)

### 4.3.2 建立DNN网络

DNN网络建立过程需要根据结果不断地修改，才能获得较为理想的准确度。在建立网络初期根据要预测的数据规模，由经验给出DNN网络参数。目标类型个数指定为两种，即恶意行为与正常行为两种标签。

1. # 设定37个特征
2. feature\_columns = [tf.contrib.layers.real\_valued\_column("", dimension=37)]
3. # 建立40，80，40的三层深度神经网络
4. classifier = tf.contrib.learn.DNNClassifier(
5. feature\_columns=feature\_columns,#把之前创建的特征列传入
6. hidden\_units=[40, 80, 40], #每层神经元数量，跟 DNN 原理有关。
7. n\_classes=2, #目标的类型的个数，目前是 2 个
8. model\_dir=MODLE\_FOLDER) #训练模型保存的路径

### 4.3.3 进行训练建立模型

训练需要指定迭代次数，次数过多越准确，但是时间开销越大，在一定程度上再增加时间对准确度已无太大影响。因此训练次数最初指定为200，之后按照20一段进行增加，直到再增加次数对准确度已无多大影响时停止。

1. #开始训练
2. classifier.fit(input\_fn=get\_train\_inputs, steps=350) #训练次数待调节
4. **print**("Training completed")

### 4.3.4 根据训练好的模型对测试集进行预测，根据结果改进模型

测试时只需要调用现有模型，使用库函数classifier.evaluate（）进行测试即可。根据测试的结果不停改进模型参数最终得到预测较为准确的模型。准确度预测代码如下：

1. #测试准确度
2. **print**("Test Strating")
3. accuracy\_score = classifier.evaluate(input\_fn=get\_test\_inputs, steps=1)[ "accuracy"]
4. **print**("nTest Accuracy: {0:f}n".format(accuracy\_score))
5. 实验结果

## 5.1 细粒度识别结果

用训练集得出的模型对测试集进行预测，根据得到的准确度修改模型，当预测的准确的达到一定程度即停止对模型的修改。最终的到准确度为99%的模型。程序截图见附录7.2

在实验过程中遇到的一个问题是迭代次数的选择，为此我们进行了测试，不断提高训练集迭代次数，观察其在预测测试集时准确度的变化，具体结果如下：

Epoch 000: Loss: 5.640, Accuracy: 94.533%

Epoch 001: Loss: 0.030, Accuracy: 99.333%

Epoch 002: Loss: 0.023, Accuracy: 99.500%

Epoch 003: Loss: 0.020, Accuracy: 99.533%

Epoch 004: Loss: 0.018, Accuracy: 99.567%

Epoch 005: Loss: 0.016, Accuracy: 99.600%

Test set accuracy: 99.526%

[Finished in 53.0s]

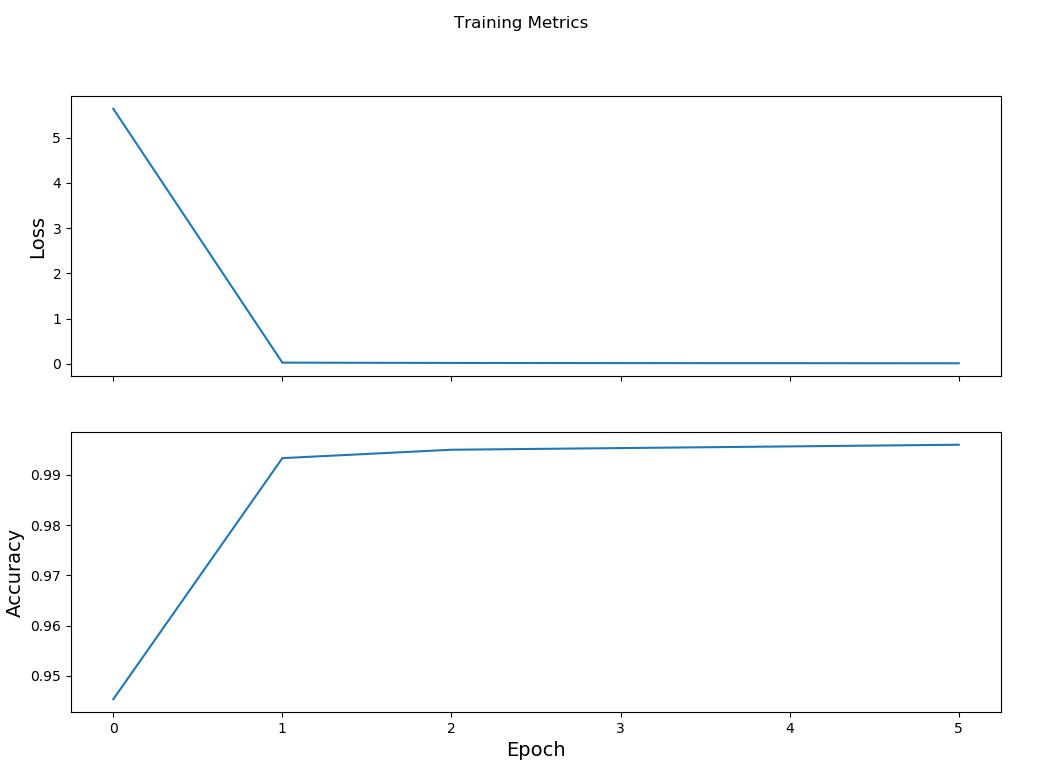


图5.1 细粒度识别迭代次数与准确度关系

注：图中Accuracy表示准确度计算值，Loss表示损失函数计算值，Epoch表示迭代次数

从实验结果中，我们可以看出，损失函数计算值随着迭代次数的增加呈现一个下降的趋势并逐渐趋向于0；准确度函数计算值随着迭代次数的增加逐渐增加，并逐渐趋向于100%。

由于随着迭代次数的增加，时间开销愈加庞大，但是准确度的提升并不明显，于是本方案在准确度不随着迭代次数明显增加时即停止迭代。

## 5.2 粗粒度识别结果

粗粒度识别的准确度在91%~96%之间波动，经过多次尝试我们得到了识别准确度为95.035%的模型。粗粒度识别的准确度相对于细粒度识别要低。程序运行结果截图见附录7.3。

1. 总结

经过上述步骤，细粒度识别准确度最高为99%，粗粒度识别准确度最高为95%。得到了较为准确的训练模型，在得到一条访问条目后，先使用细粒度识别进行检测，如果测试为正常行为，接着用粗粒度模型进行检测，如果还预测为正常行为，则此访问为正常行为。由此我们得到了一个较为准确的基于Tensorflow的有监督恶意行为特征识别方案。

通过本项目，我们了解了Tensorflow架构的基本操作，学习了机器学习的相关知识。我们也意识到任何实验都不能轻视，最初认为会一帆风顺的实验也许会得出极低的准确度。从DNN到顺序模型，不停的调节参数，修改模型，几经波折终于得出一个较为准确的预测方案。当最终结果出来时，我们觉得，一切的努力都是值得的。

1. 附录

## 7.1 KDD cup 1999 数据特征描述

**1. TCP连接基本特征（共9种）**

* 1. duration. 连接持续时间，以秒为单位，连续类型。范围是 [0, 58329] 。
  2. protocol\_type. 协议类型，离散类型，共有3种：TCP, UDP, ICMP。
  3. service. 目标主机的网络服务类型，离散类型，共有70种。
  4. flag. 连接正常或错误的状态，离散类型，共11种。
  5. src\_bytes. 从源主机到目标主机的数据的字节数，连续类型。
  6. dst\_bytes. 从目标主机到源主机的数据的字节数，连续类型。
  7. land. 若连接来自/送达同一个主机/端口则为1，否则为0，离散类型，0或1。
  8. wrong\_fragment. 错误分段的数量，连续类型，范围是 [0, 3]。
  9. urgent. 加急包的个数，连续类型，范围是[0, 14]。

**2. TCP连接的内容特征（共13种）**

（10）hot. 访问系统敏感文件和目录的次数，连续，范围是 [0, 101]。

（11）num\_failed\_logins. 登录尝试失败的次数。连续，[0, 5]。

（12）logged\_in. 成功登录则为1，否则为0，离散，0或1。

（13）num\_compromised. compromised条件（\*\*）出现的次数，连续，[0, 7479]。

（14）root\_shell. 若获得root shell 则为1，否则为0，离散，0或1。root\_shell是指获得超级用户权限。

（15）su\_attempted. 若出现”su root” 命令则为1，否则为0，离散，0或1。

（16）num\_root. root用户访问次数，连续，[0, 7468]。

（17）num\_file\_creations. 文件创建操作的次数，连续，[0, 100]。

（18）num\_shells. 使用shell命令的次数，连续，[0, 5]。

（19）num\_access\_files. 访问控制文件的次数，连续，[0, 9]。

（20）num\_outbound\_cmds. 一个FTP会话中出站连接的次数，连续。

（21）is\_hot\_login.登录是否属于“hot”列表，是为1，否则为0，例如超级用户或管理员登录。

（22）is\_guest\_login. 若是guest 登录则为1，否则为0，离散，0或1。

**3. 基于时间的网络流量统计特征 （共9种，23～31）**

（23）count. 过去两秒内，与当前连接具有相同的目标主机的连接数，连续，[0, 511]。

（24）srv\_count. 过去两秒内，与当前连接具有相同服务的连接数，连续，[0, 511]。

（25）serror\_rate. 过去两秒内，在与当前连接具有相同目标主机的连接中，出现“SYN” 错误的连接的百分比，连续，[0.00, 1.00]。

（26）srv\_serror\_rate. 过去两秒内，在与当前连接具有相同服务的连接中，出现“SYN” 错误的连接的百分比，连续，[0.00, 1.00]。

（27）rerror\_rate. 过去两秒内，在与当前连接具有相同目标主机的连接中，出现“REJ” 错误的连接的百分比，连续，[0.00, 1.00]。

（28）srv\_rerror\_rate. 过去两秒内，在与当前连接具有相同服务的连接中，出现“REJ” 错误的连接的百分比，连续，[0.00, 1.00]。

（29）same\_srv\_rate. 过去两秒内，在与当前连接具有相同目标主机的连接中，与当前连接具有相同服务的连接的百分比，连续，[0.00, 1.00]。

（30）diff\_srv\_rate. 过去两秒内，在与当前连接具有相同目标主机的连接中，与当前连接具有不同服务的连接的百分比，连续，[0.00, 1.00]。

（31）srv\_diff\_host\_rate. 过去两秒内，在与当前连接具有相同服务的连接中，与当前连接具有不同目标主机的连接的百分比，连续，[0.00, 1.00]。

注：这一大类特征中，23、25、27、29、30这5个特征是 “same host” 特征，前提都是与当前连接具有相同目标主机的连接；24、26、28、31这4个特征是 “same service” 特征，前提都是与当前连接具有相同服务的连接。

**5. 基于主机的网络流量统计特征 （共10种，32～41）**

（32）dst\_host\_count. 前100个连接中，与当前连接具有相同目标主机的连接数，连续，[0, 255]。

（33）dst\_host\_srv\_count. 前100个连接中，与当前连接具有相同目标主机相同服务的连接数，连续，[0, 255]。

（34）dst\_host\_same\_srv\_rate. 前100个连接中，与当前连接具有相同目标主机相同服务的连接所占的百分比，连续，[0.00, 1.00]。

（35）dst\_host\_diff\_srv\_rate. 前100个连接中，与当前连接具有相同目标主机不同服务的连接所占的百分比，连续，[0.00, 1.00]。

（36）dst\_host\_same\_src\_port\_rate. 前100个连接中，与当前连接具有相同目标主机相同源端口的连接所占的百分比，连续，[0.00, 1.00]。

（37）dst\_host\_srv\_diff\_host\_rate. 前100个连接中，与当前连接具有相同目标主机相同服务的连接中，与当前连接具有不同源主机的连接所占的百分比，连续，[0.00, 1.00]。

（38）dst\_host\_serror\_rate. 前100个连接中，与当前连接具有相同目标主机的连接中，出现SYN错误的连接所占的百分比，连续，[0.00, 1.00]。

（39）dst\_host\_srv\_serror\_rate. 前100个连接中，与当前连接具有相同目标主机相同服务的连接中，出现SYN错误的连接所占的百分比，连续，[0.00, 1.00]。

（40）dst\_host\_rerror\_rate. 前100个连接中，与当前连接具有相同目标主机的连接中，出现REJ错误的连接所占的百分比，连续，[0.00, 1.00]。

（41）dst\_host\_srv\_rerror\_rate. 前100个连接中，与当前连接具有相同目标主机相同服务的连接中，出现REJ错误的连接所占的百分比，连续，[0.00, 1.00]。

## 7.2 细粒度识别运行结果截图

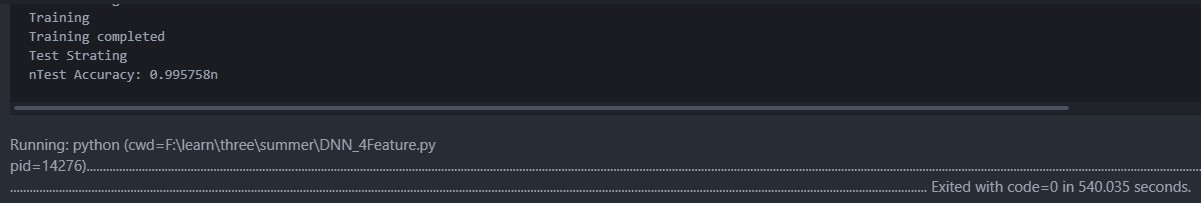


图7.1 细粒度识别准确度

由于细粒度识别较为准确一直在99%附近，由此不再分上下限。

## 7.3 粗粒度识别运行结果截图

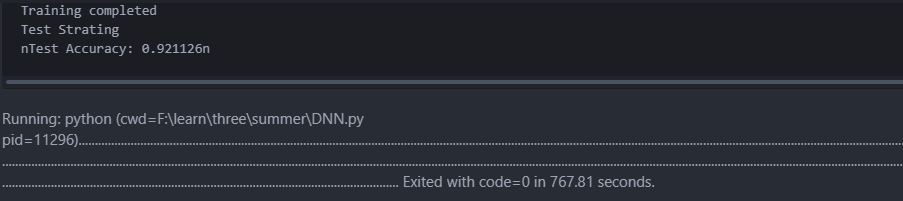
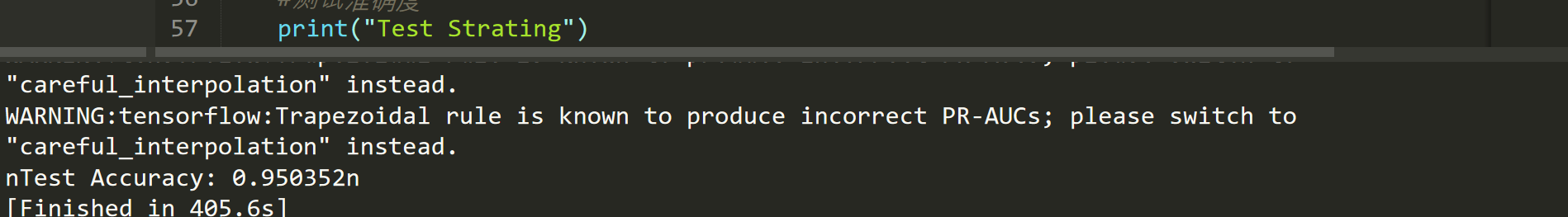


图7.2 粗粒度识别准确度下限

图7.3 粗粒度识别准确度上限