

(a) Generator Architecture G

1. Input Noise: The Generator starts with a random noise vector, which acts as a seed for generating realistic samples. This noise vector helps the Generator create diverse outputs by allowing different random variations.

2. Linear Layer and Reshape:

• The input noise is first passed through a linear layer. This transforms the low-dimensional noise vector into a higher-dimensional representation, preparing it for the subsequent layers.

• After the linear layer, the output is reshaped to fit the dimensions required by the convolutional layers.

3. Residual Blocks:

• The core of the Generator consists of a series of Residual Blocks. Residual blocks are often used in deep networks to improve gradient flow and make the network easier to train by allowing a “shortcut” path for gradients.

• Each residual block refines the generated sample, gradually transforming the noise into a realistic data representation. Residual connections (or skip connections) allow the Generator to learn both low-level and high-level features effectively.

• The Generator here has five residual blocks (Residual Block 1 to Residual Block 5), which progressively transform the data.

4. 1D Convolution Layer:

• After the residual blocks, a 1D convolutional layer is applied to reduce the dimensions of the data to match the target output format.

• This convolution layer helps refine the output, adding fine details to make it more realistic.

5. Transpose Operation and Softmax:

• A transpose operation is applied to format the data as needed (e.g., changing the order of dimensions).

• Finally, a softmax activation is applied, which converts the output into a probability distribution. This is suitable for models generating sequential data like passwords, where each output token (e.g., character) has a probability of being chosen.

6. Output:

• The final output represents a generated sample (like a password) that the Generator believes could pass as real data.

(b) Discriminator Architecture D

1. Input:

• The Discriminator receives either real data or generated data as input. Its job is to distinguish between the two by evaluating the realism of the input sample.

2. Transpose Operation and 1D Convolution Layer:

• Similar to the Generator, the Discriminator starts by applying a transpose operation followed by a 1D convolution layer. This layer transforms the input into a form suitable for feature extraction by the following residual blocks.

3. Residual Blocks:

• The Discriminator also consists of a series of residual blocks (Residual Block 1 to Residual Block 5). These blocks enable the Discriminator to extract hierarchical features from the input data.

• As in the Generator, these residual blocks help maintain gradient flow, which is particularly beneficial in GANs where training can be unstable.

4. Reshape and Linear Layer:

• After the residual blocks, the data is reshaped as needed, followed by a linear layer that reduces the output dimensions to a single value representing the “realness” of the sample.

5. Output:

• The final output of the Discriminator is a single value, often passed through a sigmoid activation function (not shown here but commonly used), representing the probability that the input is real.

• This output allows the Discriminator to “score” each sample, enabling it to distinguish real samples from generated ones.

Summary of Principles

• Generator G : Transforms random noise into realistic samples (e.g., passwords) by progressively refining the data through residual blocks and convolution layers.

• Discriminator D : Evaluates the realism of samples, either accepting them as real or rejecting them as fake, by extracting features through residual blocks and using a final linear layer to produce a probability.

Purpose of the Architecture

In this GAN setup:

• The Generator and Discriminator train in a competitive process where the Generator aims to produce realistic samples to fool the Discriminator.

• The Discriminator, on the other hand, learns to improve its accuracy in distinguishing real from fake samples.

• Over time, the Generator becomes better at generating realistic samples, while the Discriminator becomes more skilled at recognizing fake ones, leading to a balanced and effective model.

Generator Code

**def** Generator(n\_samples, seq\_len, layer\_dim, output\_dim, prev\_outputs=**None**):

output = make\_noise(shape=[n\_samples, 128])

output = lib.ops.linear.Linear('Generator.Input', 128, seq\_len \* layer\_dim, output)

output = tf.reshape(output, [-1, layer\_dim, seq\_len])

# print('Generator.1', output, layer\_dim)

output = ResBlock('Generator.1', output, layer\_dim)

output = ResBlock('Generator.2', output, layer\_dim)

output = ResBlock('Generator.3', output, layer\_dim)

output = ResBlock('Generator.4', output, layer\_dim)

output = ResBlock('Generator.5', output, layer\_dim)

output = lib.ops.conv1d.Conv1D('Generator.Output', layer\_dim, output\_dim, 1, output)

output = tf.transpose(output, [0, 2, 1])

output = softmax(output, output\_dim)

**return** output

**Discriminator Code**

**def** Discriminator(inputs, seq\_len, layer\_dim, input\_dim):

output = tf.transpose(inputs, [0,2,1])

output = lib.ops.conv1d.Conv1D('Discriminator.Input', input\_dim, layer\_dim, 1, output)

output = ResBlock('Discriminator.1', output, layer\_dim)

output = ResBlock('Discriminator.2', output, layer\_dim)

output = ResBlock('Discriminator.3', output, layer\_dim)

output = ResBlock('Discriminator.4', output, layer\_dim)

output = ResBlock('Discriminator.5', output, layer\_dim)

output = tf.reshape(output, [-1, seq\_len \* layer\_dim])

output = lib.ops.linear.Linear('Discriminator.Output', seq\_len \* layer\_dim, 1, output)

**return** output

**Block Code**

**def** ResBlock(name, inputs, dim):

output = inputs

output = tf.nn.relu(output)

# print(name+'.1', dim, dim, 5, output)

output = lib.ops.conv1d.Conv1D(name+'.1', dim, dim, 5, output)

output = tf.nn.relu(output)

output = lib.ops.conv1d.Conv1D(name+'.2', dim, dim, 5, output)

**return** inputs + (0.3\*output)

**Noise Code**

**def** make\_noise(shape):

**return** tf.random.normal(shape)

**Conv1D Code**

**def** Conv1D(name, input\_dim, output\_dim, filter\_size, inputs, he\_init=**True**, mask\_type=**None**, stride=1, weightnorm=**None**, biases=**True**, gain=1):

"""

inputs: tensor of shape (batch size, num channels, width)

mask\_type: one of None, 'a', 'b'

returns: tensor of shape (batch size, num channels, width)

"""

**with** tf.name\_scope(name) **as** scope:

**if** mask\_type **is** **not** **None**:

mask\_type, mask\_n\_channels = mask\_type

mask = np.ones(

(filter\_size, input\_dim, output\_dim),

dtype='float32'

)

center = filter\_size // 2

# Mask out future locations

# filter shape is (width, input channels, output channels)

mask[center+1:, :, :] = 0.

# Mask out future channels

**for** i **in** range(mask\_n\_channels):

**for** j **in** range(mask\_n\_channels):

**if** (mask\_type=='a' **and** i >= j) **or** (mask\_type=='b' **and** i > j):

mask[

center,

i::mask\_n\_channels,

j::mask\_n\_channels

] = 0.

**def** uniform(stdev, size):

**return** np.random.uniform(

low=-stdev \* np.sqrt(3),

high=stdev \* np.sqrt(3),

size=size

).astype('float32')

fan\_in = input\_dim \* filter\_size

fan\_out = output\_dim \* filter\_size / stride

**if** mask\_type **is** **not** **None**: # only approximately correct

fan\_in /= 2.

fan\_out /= 2.

**if** he\_init:

filters\_stdev = np.sqrt(4./(fan\_in+fan\_out))

**else**: # Normalized init (Glorot & Bengio)

filters\_stdev = np.sqrt(2./(fan\_in+fan\_out))

filter\_values = uniform(

filters\_stdev,

(filter\_size, input\_dim, output\_dim)

)

# print "WARNING IGNORING GAIN"

filter\_values \*= gain

filters = lib.param(name+'.Filters', filter\_values)

**if** weightnorm==**None**:

weightnorm = \_default\_weightnorm

**if** weightnorm:

norm\_values = np.sqrt(np.sum(np.square(filter\_values), axis=(0,1)))

target\_norms = lib.param(

name + '.g',

norm\_values

)

**with** tf.name\_scope('weightnorm') **as** scope:

norms = tf.sqrt(tf.reduce\_sum(tf.square(filters), reduction\_indices=[0,1]))

filters = filters \* (target\_norms / norms)

**if** mask\_type **is** **not** **None**:

**with** tf.name\_scope('filter\_mask'):

filters = filters \* mask

result = tf.nn.conv1d(

input=inputs,

filters=filters,

stride=stride,

padding='SAME',

data\_format='NCW'

)

**if** biases:

\_biases = lib.param(

name+'.Biases',

np.zeros([output\_dim], dtype='float32')

)

# result = result + \_biases

result = tf.expand\_dims(result, 3)

result = tf.nn.bias\_add(result, \_biases, data\_format='NCW')

result = tf.squeeze(result)

**return** result

(a) 生成器架构 G

1. 输入噪声（Input Noise）：

• 生成器的输入是一个随机噪声向量，它充当生成真实样本（如密码）的种子。

• 这个噪声向量可以提供随机性，使生成器可以生成不同的样本，从而保证多样性。

2. 线性层（Linear Layer）和重塑（Reshape）：

• 噪声向量首先经过一个线性层，将低维度的噪声转化为高维度的表示，为后续的层准备合适的输入。

• 然后，对输出进行重塑，调整为后续卷积层所需的形状。

3. 残差块（Residual Blocks）：

• 生成器的核心部分是多个残差块。残差块用于在深层神经网络中改善梯度传播，使训练更稳定。

• 每个残差块都会细化生成的样本，将噪声逐步转换为逼真的数据表示。残差连接（跳跃连接）可以帮助生成器学习到不同层次的特征。

• 在图中，生成器包含了 5 个残差块（Residual Block 1 到 Residual Block 5），它们会逐步变换数据，增加生成的细节和真实性。

4. 一维卷积层（Conv 1D）：

• 在残差块之后，经过一个一维卷积层，进一步调整数据的维度，使输出的维度符合目标格式。

• 卷积层帮助生成器进一步调整输出细节，以接近真实样本。

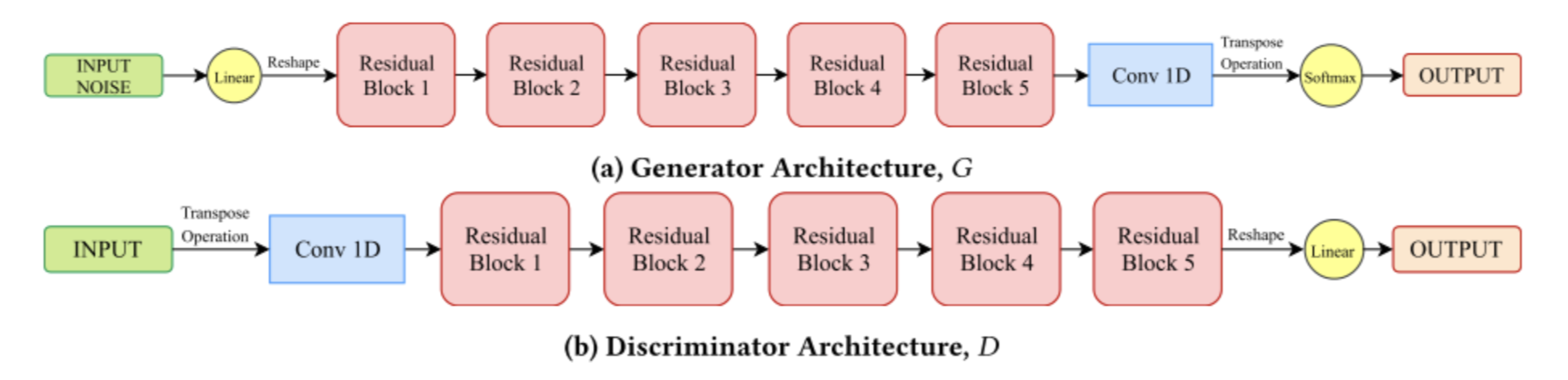
5. 转置操作和 Softmax 激活函数：

• 转置操作用于调整数据的形状，适应后续的处理。

• Softmax 激活函数将输出转化为概率分布，这对于生成密码等序列化数据特别有用，每个字符或符号都有一个被选中的概率。

6. 输出（Output）：

• 最终输出是生成器生成的一个样本（例如，一个密码），生成器尝试生成出真实的数据来“欺骗”判别器。



(b) 判别器架构 D

1. 输入（Input）：

• 判别器接收输入数据，可以是真实的数据，也可以是生成器生成的数据。判别器的任务是判断输入的真实性。

2. 转置操作和一维卷积层：

• 判别器首先进行转置操作，接着通过一个一维卷积层将输入转换为适合特征提取的形式。

• 这层卷积有助于提取输入数据的低层次特征，为后续的残差块处理提供基础。

3. 残差块（Residual Blocks）：

• 判别器中同样包含一系列残差块（Residual Block 1 到 Residual Block 5），这些残差块可以帮助判别器提取层次化的特征，提升对样本真实性的判断能力。

• 和生成器类似，残差块可以让模型学习到更加丰富的特征，保持梯度流动，提高模型稳定性。

4. 重塑（Reshape）和线性层（Linear Layer）：

• 经过残差块后，数据会被重塑为适合线性层的形状，然后通过一个线性层输出一个单一数值，用于表示该样本的真实性。

5. 输出（Output）：

• 判别器的最终输出是一个概率值，通常使用 sigmoid 激活函数（图中未显示）将输出值限制在 [0, 3] 之间，表示输入数据是真实的概率。

• 这个输出帮助判别器对每个输入样本进行评分，从而实现识别真假数据的目的。

总结原理

在这个 GAN 架构中：

• 生成器 G ：将随机噪声转化为逼真的数据（例如密码），通过残差块和卷积层逐步细化数据，使输出更接近真实。

• 判别器 D ：判别器接收输入数据并通过残差块提取特征，然后输出一个概率，表示输入数据的真实性。

架构的目的

在训练过程中：

• 生成器和判别器通过对抗训练的方式相互改进。生成器尝试生成更逼真的样本，以骗过判别器；

• 而判别器则在训练中不断提升自己，以识别出生成的样本与真实数据的差异。

• 这种相互竞争的训练方式让生成器生成的样本越来越逼真，而判别器也更准确地判断真假，最终实现一个平衡和稳定的模型。

PassGAN 训练模型主要使用了 RockYou 数据集。以下是对该数据集的详细说明：

1. 数据集概览：

• 名称： RockYou 数据集

• 大小： 约 133 MB

• 条目数： 包含约 3,200 万条密码记录

2. 数据预处理：

在用于训练前，数据集通常会经过以下处理步骤：

• 清洗： 移除重复项和空白行

• 过滤： 删除过短或过长的密码，以确保模型训练的有效性

• 编码： 将密码字符转换为模型可处理的数值形式

3. 密码长度分布：

RockYou 数据集的密码长度分布多样，常见长度在 6 到 12 个字符之间。具体的长度分布统计可通过分析数据集获得。

4. 训练/验证集划分比例：

在深度学习模型训练中，通常将数据集按 80% 用于训练，20% 用于验证的比例进行划分。然而，PassGAN 项目中并未明确说明其具体的划分比例。