11.Building A Neural Network Malware Detector With Keras

BUNU MACHINE UYARLAMAYA ÇALIŞaBİLRİM

On yıl önce, işleyen, ölçeklenebilir ve hızlı bir sinir ağı oluşturmak zaman alıcıydı ve oldukça fazla kod gerektiriyordu. Bununla birlikte, son birkaç yılda, sinir ağı tasarımına yönelik yüksek seviyeli arayüzler geliştirildiğinden, bu süreç çok daha az acı verici hale geldi. Python paketi Keras bu arayüzlerden biridir.

Bu bölümde, Keras paketini kullanarak örnek bir sinir ağının nasıl oluşturulacağını size göstereceğim. İlk olarak, Keras'ta bir modelin mimarisinin nasıl tanımlanacağını açıklayacağım. İkinci olarak, bu modeli iyi huylu ve kötü niyetli HTML dosyalarını ayırt etmek için eğitiyoruz ve bu tür modelleri nasıl kaydedip yükleyeceğinizi öğreniyorsunuz. Üçüncüsü, Python paketi sklearn'ı kullanarak, modelin doğrulama verileri üzerindeki doğruluğunu nasıl değerlendireceğinizi öğrenirsiniz. Son olarak, doğrulama doğruluğu raporlamasını model eğitim sürecine entegre etmek için öğrendiklerinizi kullanırız.

Bu kitapla birlikte verilen verilerdeki ilgili kodu okurken ve düzenlerken bu bölümü okumanızı tavsiye ederim. Bu bölümde tartışılan tüm kodu (işleri çalıştırmayı ve ayarlamayı kolaylaştırmak için parametreleştirilmiş işlevler halinde düzenlenmiş) ve birkaç ekstra örnek burada bulabilirsiniz. Bu bölümün sonunda, kendinizi kendi ağlarınızı kurmaya hazır hissedeceksiniz!

Bu bölümdeki kod listelerini çalıştırmak için, yalnızca bu bölümün ch11 / requirements.txt dosyasında (pip install -r requirements.txt) listelenen paketleri kurmanız değil, aynı zamanda Keras'ın arka uç motorlarından birini bilgisayarınıza kurmak için yönergeleri de izlemeniz gerekir. sistemi (TensorFlow, Theano veya CNTK). Buradaki talimatları izleyerek TensorFlow'u kurun: https://www.tensorflow.org/ install /.

\*Defining a Model’s Architecture: Bir Modelin Mimarisini Tanımlama

Bir sinir ağı oluşturmak için, mimarisini tanımlamanız gerekir: hangi nöronlar nereye gidecek, sonraki nöronlara nasıl bağlanacaklar ve verilerin her şeyde nasıl aktığı. Neyse ki Keras, tüm bunları tanımlamak için basit, esnek bir arayüz sağlar. Keras aslında model tanımı için iki benzer sözdizimini destekler, ancak diğer ("sıralı") sözdiziminden daha esnek ve güçlü olduğu için İşlevsel API sözdizimini kullanacağız.

      Bir model tasarlarken üç şeye ihtiyacınız vardır: girdi, ortadaki girdiyi işleyen şeyler ve çıktı. Bazen modelleriniz birden çok girdiye, birden çok çıktıya ve ortada çok karmaşık şeylere sahip olabilir, ancak temel fikir, bir modelin mimarisini tanımlarken, yalnızca girdiyi - HTML ile ilgili özellikler gibi verilerinizi - tanımladığınızdır. dosya — çeşitli nöronlardan (ortadaki şeyler) akar ve sonunda son nöronlar bir miktar çıktı üretene kadar.

Bu mimariyi tanımlamak için Keras katmanları kullanır. Bir katman, hepsi aynı tür aktivasyon işlevini kullanan, tümü önceki bir katmandan veri alan ve çıktılarını sonraki bir nöron katmanına gönderen bir grup nörondur. Bir sinir ağında, giriş verileri genellikle ilk nöron katmanına beslenir; bu katman çıktılarını bir sonraki katmana gönderir, çıktılarını başka bir katmana gönderir ve bu böyle devam eder, son nöron katmanı ağın son çıktı.

     11-1 Listesi, Keras’ın işlevsel API sözdizimi kullanılarak tanımlanan basit bir model örneğidir. Kodda satır satır ilerlerken kodu kendiniz yazıp çalıştırmak için yeni bir Python dosyası açmanızı tavsiye ederim. Alternatif olarak, ch11 / model\_architecture.py dosyasının bölümlerini bir ipython oturumuna kopyalayıp yapıştırarak veya bir terminal penceresinde python ch11 / model\_architecture.py çalıştırarak bu kitapla birlikte verilen verilerdeki ilişkili kodu çalıştırmayı deneyebilirsiniz.

(1) from keras import layers

(2) from keras.models import Model

input = layers.Input((3)shape=(1024,), (4)dtype='float32')

(5) middle = layers.Dense(units=512, activation='relu')(input)

(6) output = layers.Dense(units=1, activation='sigmoid')(middle)

(7) model = Model(inputs=input, outputs=output)

model.compile((8)optimizer='adam',

(9)loss='binary\_crossentropy',

(10)metrics=['accuracy'])

Listing 11-1: Defining a simple model using functional API syntax:Liste 11-1: İşlevsel API sözdizimini kullanarak basit bir model tanımlama

İlk olarak, Keras paketinin katman alt modülünü (1) ve Keras'ın models alt modülünden (2) Model sınıfını içe aktarıyoruz.

Daha sonra, katmanlara bir şekil değeri (a tuple of integers) (3) ve bir veri türü (dize string) (4) ileterek bu modelin bir gözlem için ne tür verileri kabul edeceğini belirtiyoruz. Burada, modelimize girdi verisinin 1.024 floats dizisi olacağını ilan ettik. Örneğin girdimiz bir tamsayı matrisi olsaydı, ilk satır daha çok input = Input (şekil = (100, 100,) dtype = 'int32') gibi görünürdü.

Not:Model, bir boyutta değişken boyutlu girdiler alırsa, sayı yerine Yok'u kullanabilirsiniz — örneğin, (100, None,).

Ardından, bu giriş verilerinin gönderileceği nöron katmanını belirtiyoruz. Bunu yapmak için, içe aktardığımız katman alt modülünü, özellikle Yoğun işlevini Dense function (4) kullanarak, bu katmanın yoğun şekilde bağlı (tamamen bağlı olarak da adlandırılır) bir katman olacağını belirtmek için tekrar kullanıyoruz; bu katmandaki her nörona. Yoğun, Keras modellerini geliştirirken kullanacağınız en yaygın katman türüdür. Diğerleri, verilerin şeklini değiştirme (Yeniden Şekillendirme Reshape) ve kendi özel katmanınızı (Lambda) uygulama gibi şeyler yapmanıza izin verir.

Yoğun fonksiyonuna iki argüman aktarıyoruz: bu katmanda 512 nöron istediğimizi belirtmek için birim = 512 ve bu nöronların düzeltilmiş doğrusal birim (ReLU) nöronlar olmasını istediğimizi belirtmek için aktivasyon = 'relu'. (Bölüm 10'dan ReLU nöronlarının, hangisi daha büyükse çıktı veren basit bir aktivasyon işlevi kullandığını hatırlayın: 0 veya nöronun girdilerinin ağırlıklı toplamı.) Katmanları kullanıyoruz. Dense (birimler = 512, aktivasyon = 'relu' units=512, activation='relu') katmanı tanımlamak için ve ardından satırın son kısmı - (girdi) bu katmana girişi (yani bizim girdi nesnemiz) bildirir. Katmanımıza bu giriş geçişinin, kod satırlarının sıralanmasının aksine, modelde veri akışının nasıl tanımlandığını anlamak önemlidir.

       Sonraki satırda, yine Yoğun işlevini Dense function kullanan modelimizin çıktı katmanını tanımlıyoruz. Ancak bu sefer katmana yalnızca tek bir nöron atıyoruz ve bir 'sigmoid' aktivasyon işlevi (6) kullanıyoruz; bu, çok sayıda veriyi 0 ile 1 arasındaki tek bir skorda birleştirmek için harikadır. Çıkış katmanı, (ortadaki ) girdi olarak nesne, orta katmanımızda ki 512 nöronumuzun çıktılarının hepsinin bu nörona gönderilmesi gerektiğini bildirir.

Artık katmanlarımızı tanımladığımıza göre, tüm bu katmanları bir model olarak bir araya getirmek için modeller alt modülündeki Model sınıfını kullanıyoruz (7). Yalnızca girdi katmanlarınızı ve çıktı katmanlarınızı belirtmeniz gerektiğini unutmayın. Birinciden sonraki her katmana önceki katman girdi olarak verildiğinden, son çıktı katmanı, modelin önceki katmanlarla ilgili ihtiyaç duyduğu tüm bilgileri içerir. Giriş ve çıkış katmanlarımız arasında 10 tane daha orta katman tanımlayabiliriz, ancak (7) 'deki kod satırı aynı kalacaktır.

\*Compiling the Model:Modeli Derlemek

Son olarak, modelimizi derlememiz gerekiyor. Modelin mimarisini ve veri akışını tanımladık, ancak modelin eğitimini nasıl gerçekleştirmesini istediğimizi henüz belirtmedik. Bunu yapmak için, modelimizin kendi derleme yöntemini compile method  kullanıyoruz ve ona üç parametre aktarıyoruz:

-İlk parametre olan optimizer (8), kullanılacak geri yayılım algoritmasının türünü belirtir. Kullanmak istediğiniz algoritmanın adını burada yaptığımız gibi bir karakter dizisi aracılığıyla belirleyebilir veya doğrudan keras.optimizers'dan bir algoritmayı içe aktararak belirli parametreleri algoritmaya aktarabilir veya hatta kendi algoritmanızı tasarlayabilirsiniz.

-Kayıp parametresi (9), eğitim süreci (geri yayılım) sırasında en aza indirilen şeyi belirtir. Özellikle bu, gerçek eğitim etiketleriniz ile modelinizin öngörülen etiketleri (çıktı) arasındaki farkı temsil etmek için kullanmak istediğiniz formülü belirtir. Yine, bir kayıp işlevinin adını belirtebilir veya keras.losses.mean\_squared\_error gibi gerçek bir işlevi iletebilirsiniz.

-Son olarak, metrik parametresi (10) için, eğitim sırasında ve sonrasında model performansını analiz ederken Keras'ın raporlamasını istediğiniz metriklerin bir listesini iletebilirsiniz. Yine, ['categorical\_accuracy', keras.metrics.top\_k\_categorical\_accur cy] gibi dizeleri veya gerçek metrik işlevleri iletebilirsiniz.

Kodu Liste 11-1'de çalıştırdıktan sonra, ekranınıza yazdırılan model yapısını görmek için model.summary () öğesini çalıştırın. Çıktınız Şekil 11-1'deki gibi görünmelidir.

Figure 11-1: Output of model.summary()

Şekil 11-1 model.summary () çıktısını gösterir. Her katmanın açıklaması, o katmanla ilişkili parametrelerin sayısıyla birlikte ekrana yazdırılır. Örneğin, yoğun\_1 katmanı 524.800 parametreye sahiptir çünkü 512 nöronunun her biri giriş katmanından 1.024 giriş değerinin bir kopyasını alır, bu da 1.024 × 512 ağırlık olduğu anlamına gelir. 512 önyargı parametresi eklediğinizde 1.024 × 512 + 512 = 524.800 elde edersiniz. Modelimizi henüz eğitmemiş veya doğrulama verileri üzerinde test etmemiş olsak da bu, eğitilmeye hazır derlenmiş bir Keras modelidir!

NOT:Biraz daha karmaşık bir model örneği için ch11 / model\_architecture.py'deki örnek kodu inceleyin!

\*Training the Model:Modeli Eğitmek

Modelimizi eğitmek için eğitim verilerine ihtiyacımız var. Bu kitapla birlikte gelen sanal makine, yaklaşık yarım milyon iyi huylu ve kötü niyetli HTML dosyası içerir. Bu, zararsız (ch11 / data / html / benign\_files /) ve kötü amaçlı (ch11 / data / html / malware\_files /) HTML dosyalarının bulunduğu iki klasörden oluşur. (Bu dosyaları bir tarayıcıda açmamayı unutmayın!) Bu bölümde, bunları sinir ağımızı bir HTML dosyasının iyi huylu (0) veya kötü niyetli (1) olup olmadığını tahmin etmek üzere eğitmek için kullanırız.

\*Extracting Features:Unsurları Çıkarma

Bunu yapmak için önce verilerimizi nasıl temsil edeceğimize karar vermemiz gerekiyor. Başka bir deyişle, modelimize girdi olarak kullanmak için her HTML dosyasından hangi özellikleri çıkarmak istiyoruz? Örneğin, her bir HTML dosyasındaki ilk 1.000 karakteri modele iletebiliriz, alfabedeki tüm harflerin sıklık sayılarını aktarabiliriz veya daha karmaşık özellikler geliştirmek için bir HTML ayrıştırıcı kullanabiliriz. İşleri kolaylaştırmak için, değişken uzunluklu, potansiyel olarak çok büyük olan her HTML dosyasını, modelimizin önemli kalıpları hızlı bir şekilde işlemesine ve öğrenmesine olanak tanıyan tek tip boyutlu, sıkıştırılmış bir gösterime dönüştüreceğiz.

Bu örnekte, her bir HTML dosyasını 1.024 uzunlukta bir kategori sayıları vektörüne dönüştürüyoruz; burada her kategori sayısı, HTML dosyasındaki hash değeri verilen kategoriye çözümlenen simge sayısını temsil eder. 11-2 listesi, özellik çıkarma kodunu gösterir.

import numpy as np

import murmur

import re

import os

def read\_file(sha, dir):

with open(os.path.join(dir, sha), 'r') as fp:

file = fp.read()

return file

def extract\_features(sha, path\_to\_files\_dir,

hash\_dim=1024, (1)split\_regex=r"\s+"):

(2) file = read\_file(sha=sha, dir=path\_to\_files\_dir)

(3) tokens = re.split(pattern=split\_regex, string=file)

# now take the modulo(hash of each token) so that each token is replaced

# by bucket (category) from 1:hash\_dim.

token\_hash\_buckets = [

(4) (murmur.string\_hash(w) % (hash\_dim - 1) + 1) for w in tokens

]

# Finally, we'll count how many hits each bucket got, so that our features

# always have length hash\_dim, regardless of the size of the HTML file:

token\_bucket\_counts = np.zeros(hash\_dim)

# this returns the frequency counts for each unique value in

# token\_hash\_buckets:

buckets, counts = np.unique(token\_hash\_buckets, return\_counts=True)

# and now we insert these counts into our token\_bucket\_counts object:

for bucket, count in zip(buckets, counts):

(5) token\_bucket\_counts[bucket] = count

return np.array(token\_bucket\_counts)

Listing 11-2: Feature extraction code:Liste 11-2: Özellik çıkarma kodu

<https://medium.com/@iratoon/bypassing-web-application-firewall-part-3-521fd66a6422>

https://drive.google.com/file/d/13OiUkYSj59M9e3Kxp\_mwaMuj1jkjlENu/view

bak buna

Keras'ın nasıl çalıştığını anlamak için bu kodun tüm ayrıntılarını anlamanız gerekmez, ancak neler olup bittiğini daha iyi anlamak için koddaki yorumları baştan sona okumanızı tavsiye ederim. Extract\_features işlevi, bir HTML dosyasını büyük bir dizge (2) olarak okuyarak başlar ve daha sonra bu dizeyi normal ifadeye (3) dayalı olarak bir dizi jetona böler. Daha sonra, her bir jetonun sayısal karması alınır ve bu karmalar, her bir karmanın (4) modulosu alınarak kategorilere ayrılır. Son özellik seti, histogram bin sayısı gibi, her kategorideki (5) karma sayısıdır. İsterseniz, ortaya çıkan belirteçleri ve özellikleri nasıl etkilediğini görmek için HTML dosyasını parçalara ayıran split\_regex (1) normal ifadesini değiştirmeyi deneyebilirsiniz.

Bunları atladıysanız veya anlamadıysanız, sorun değil: extract\_features fonksiyonumuzun girdi olarak bir HTML dosyasına giden yolu aldığını ve ardından 1.024 uzunluğunda bir özellik dizisine veya hash\_dim ne olursa olsun dönüştürdüğünü bilin.

\*Creating a Data Generator:Veri Oluşturucu Oluşturma:

Şimdi Keras modelimizin aslında bu özellikler üzerinde çalışmasına ihtiyacımız var. Önceden belleğe yüklenmiş küçük miktarlarda verilerle çalışırken, modelinizi Keras'ta eğitmek için Liste 11-3 gibi basit bir kod satırı kullanabilirsiniz.

# first you would load in my\_data and my\_labels via some means, and then:

model.fit(my\_data, my\_labels, epochs=10, batch\_size=32)

Listing 11-3: Training your model when data is already loaded into memory:Liste 11-3: Veriler belleğe zaten yüklendiğinde modelinizi eğitme

Ancak, büyük miktarda veriyle çalışmaya başladığınızda bu gerçekten yararlı değildir, çünkü tüm egzersiz verilerinizi aynı anda bilgisayarınızın belleğine sığdıramazsınız. Bunu aşmak için biraz daha karmaşık ancak daha ölçeklenebilir model.fit\_generator işlevini kullanıyoruz. Tüm eğitim verilerini bir kerede bu işleve aktarmak yerine, eğitim verilerini gruplar halinde veren bir oluşturucu iletirsiniz, böylece bilgisayarınızın RAM'i tıkanmaz.

Python üreteçleri, getiri ifadesine sahip olmaları dışında, Python işlevleri gibi çalışır. Tek bir sonuç döndürmek yerine, üreteçler çok sayıda veya sonsuz sonuç kümesi elde etmek için tekrar tekrar çağrılabilen bir nesne döndürür. 11-4 listesi, özellik çıkarma işlevimizi kullanarak kendi veri oluşturucumuzu nasıl oluşturabileceğimizi gösterir.

def my\_generator(benign\_files, malicious\_files,

path\_to\_benign\_files, path\_to\_malicious\_files,

batch\_size, features\_length=1024):

n\_samples\_per\_class = batch\_size / 2

(1) assert len(benign\_files) >= n\_samples\_per\_class

assert len(malicious\_files) >= n\_samples\_per\_class

(2) while True:

ben\_features = [

extract\_features(sha, path\_to\_files\_dir=path\_to\_benign\_files,

hash\_dim=features\_length)

for sha in np.random.choice(benign\_files, n\_samples\_per\_class,

replace=False)

]

mal\_features = [

(3) extract\_features(sha, path\_to\_files\_dir=path\_to\_malicious\_files,

hash\_dim=features\_length)

(4)for sha in np.random.choice(malicious\_files, n\_samples\_per\_class,

replace=False)

]

(5) all\_features = ben\_features + mal\_features

labels = [0 for i in range(n\_samples\_per\_class)] + [1 for i in range(

n\_samples\_per\_class)]

idx = np.random.choice(range(batch\_size), batch\_size)

(6) all\_features = np.array([np.array(all\_features[i]) for i in idx])

labels = np.array([labels[i] for i in idx])

(7) yield all\_features, labels

Listing 11-4: Writing a data generator:Liste 11-4: Veri oluşturucu yazma

İlk olarak, kod yeterli veri olup olmadığını kontrol etmek için iki assert ifadesi yapar (1). Sonra bir süreli (2) döngü içinde (böylece sonsuza kadar yinelenir), hem iyi huylu hem de kötü niyetli özellikler, rasgele bir dosya anahtarı örneği (4) seçilerek ve daha sonra extract\_features işlevimizi (3) kullanarak bu dosyalar için özellikler ayıklanarak yakalanır. . Daha sonra, zararsız ve kötü niyetli özellikler ve ilişkili etiketler (0 ve 1) birleştirilir (5) ve karıştırılır (6). Son olarak bu özellikler ve etiketler iade edilir (7).

Bir kez somutlaştırıldığında, bu oluşturucu, oluşturucunun sonraki () yöntemi her çağrıldığında modelin eğitim alması için batch\_size özellikleri ve etiketleri (yüzde 50 kötü niyetli, yüzde 50 zararsız) sağlamalıdır.

11-5 listesi, bu kitapla birlikte gelen verileri kullanarak bir eğitim verisi oluşturucunun nasıl oluşturulacağını ve oluşturucuyu modelimizin fit\_generator yöntemine geçirerek modelimizi nasıl eğiteceğimizi gösterir.

import os

batch\_size = 128

features\_length = 1024

path\_to\_training\_benign\_files = 'data/html/benign\_files/training/'

path\_to\_training\_malicious\_files = 'data/html/malicious\_files/training/'

steps\_per\_epoch = 1000 # artificially small for example-code speed!

(1)train\_benign\_files = os.listdir(path\_to\_training\_benign\_files)

(2) train\_malicious\_files = os.listdir(path\_to\_training\_malicious\_files)

# make our training data generator!

(3)training\_generator = my\_generator(

benign\_files=train\_benign\_files,

malicious\_files=train\_malicious\_files,

path\_to\_benign\_files=path\_to\_training\_benign\_files,

path\_to\_malicious\_files=path\_to\_training\_malicious\_files,

batch\_size=batch\_size,

features\_length=features\_length

)

(4) model.fit\_generator(

(5)generator=training\_generator,

(6)steps\_per\_epoch=steps\_per\_epoch,

(7)epochs=10

)

Liste 11-5: Eğitim oluşturucuyu oluşturma ve modeli eğitmek için kullanma

Neler olduğunu anlamak için bu kodu okumayı deneyin. Gerekli bir paketi içe aktardıktan ve bazı parametre değişkenlerini oluşturduktan sonra, zararsız (1) ve kötü niyetli eğitim verilerimizin (2) dosya adlarını belleğe okuruz (ancak dosyaların kendileri değil). Eğitim veri oluşturucumuzu almak için bu değerleri yeni my\_generator işlevimize (3) aktarıyoruz. Son olarak, Liste 11-1'deki modelimizi kullanarak, eğitime başlamak için modelin yerleşik fit\_generator yöntemini (4) kullanıyoruz.

Fit\_generator yöntemi üç parametre alır. Oluşturucu parametresi (5), her parti için eğitim verisi üreten veri oluşturucuyu belirtir. Eğitim sırasında parametreler, o grup için tüm eğitim gözlemlerinin sinyallerinin ortalaması alınarak grup başına bir kez güncellenir. Steps\_per\_epoch parametresi (6), modelin her dönemi işlemesini istediğimiz parti sayısını ayarlar. Sonuç olarak, modelin dönem başına gördüğü toplam gözlem sayısı batch\_size \* steps\_per\_epoch şeklindedir. Kural olarak, bir modelin dönem başına gördüğü gözlem sayısı, veri kümesi boyutuna eşit olmalıdır, ancak bu bölümde ve sanal makine örnek kodunda, kodumuzun daha hızlı çalışmasını sağlamak için steps\_per\_epoch'u azaltıyorum. Epochs parametresi (7) çalıştırmak istediğimiz epoch sayısını belirler.

Bu kodu, bu kitaba eşlik eden ch11 / dizininde çalıştırmayı deneyin. Bilgisayarınızın gücüne bağlı olarak, her eğitim döneminin çalışması belirli bir süre alacaktır. Etkileşimli bir oturum kullanıyorsanız, biraz zaman alıyorsa işlemi birkaç dönem sonra iptal etmekten çekinmeyin (ctrl-C). Bu, ilerlemeyi kaybetmeden eğitimi durduracaktır. Süreci iptal ettikten sonra (veya kod tamamlandıktan) sonra eğitimli bir modeliniz olacak! Sanal makine ekranınızdaki okuma, Şekil 11-2 gibi görünmelidir.

Şekil 11-2: Bir Keras modelinin eğitiminden elde edilen konsol çıktısı

İlk birkaç satır, Keras'ın varsayılan arka ucu olan TensorFlow'un yüklendiğine dikkat çekiyor. Ayrıca Şekil 11-2'deki gibi bazı uyarılar da göreceksiniz; bunlar sadece eğitimin GPU'lar yerine CPU'lar üzerinde yapılacağı anlamına gelir (GPU'lar sinir ağlarını eğitmek için genellikle yaklaşık 2–20 kat daha hızlıdır, ancak bu kitabın amaçları doğrultusunda CPU tabanlı eğitim iyidir). Son olarak, her dönem için, belirli bir dönemin ne kadar daha uzun süreceğini gösteren bir ilerleme çubuğu ile dönemin kayıp ve doğruluk ölçümlerini göreceksiniz.

\*Incorporating Validation Data:Doğrulama Verilerini Dahil Etme

Önceki bölümde, ölçeklenebilir fit\_generator yöntemini kullanarak HTML dosyalarında bir Keras modelinin nasıl eğitileceğini öğrendiniz. Gördüğünüz gibi, model eğitim sırasında her çağın mevcut kayıp ve doğruluk istatistiklerini gösteren ifadeler yazdırır. Ancak, gerçekten önem verdiğiniz şey, eğitilmiş modelinizin doğrulama verileri veya daha önce hiç görmediği veriler üzerinde nasıl çalıştığıdır. Bu, modelinizin gerçek hayattaki bir üretim ortamında karşılaşacağı veri türünü daha iyi temsil eder.

Daha iyi modeller tasarlamaya ve modelinizi ne kadar süreyle eğiteceğinizi anlamaya çalışırken, eğitim doğruluğundan ziyade doğrulama doğruluğunu en üst düzeye çıkarmaya çalışmalısınız, ikincisi Şekil 11-2'de gösterilmiştir. Daha da iyisi, bir üretim ortamını daha iyi simüle etmek için eğitim verilerinden sonraki tarihlerden gelen doğrulama dosyalarını kullanmaktır.

Liste 11-6, Liste 11-4'teki my\_generator işlevimizi kullanarak doğrulama özelliklerimizi belleğe nasıl yükleyeceğimizi gösterir.

import os

path\_to\_validation\_benign\_files = 'data/html/benign\_files/validation/'

path\_to\_validation\_malicious\_files = 'data/html/malicious\_files/validation/'

# get the validation keys:

val\_benign\_file\_keys = os.listdir(path\_to\_validation\_benign\_files)

val\_malicious\_file\_keys = os.listdir(path\_to\_validation\_malicious\_files)

# grab the validation data and extract the features:

(1)validation\_data = my\_generator(

benign\_files=val\_benign\_files,

malicious\_files=val\_malicious\_files,

path\_to\_benign\_files=path\_to\_validation\_benign\_files,

path\_to\_malicious\_files=path\_to\_validation\_malicious\_files,

(2) batch\_size=10000,

features\_length=features\_length

(3) ).next()

Listing 11-6: Reading validation features and labels into memory by using the

my\_generator function:Liste 11-6: my\_generator işlevini kullanarak doğrulama özelliklerini ve etiketleri belleğe okuma

Bu kod, eğitim verisi oluşturucumuzu oluşturma şeklimize çok benziyor, tek fark dosya yollarının değişmesi ve şimdi tüm doğrulama verilerini belleğe yüklemek istiyoruz. Dolayısıyla, sadece oluşturucuyu oluşturmak yerine, üzerinde doğrulamak istediğimiz dosya sayısına eşit büyük bir batch\_size (2) ile bir doğrulama veri oluşturucusu (1) oluşturuyoruz ve hemen next () (3) yöntemini sadece bir kez çağırıyoruz. .

Artık belleğe yüklenen bazı doğrulama verilerimiz olduğuna göre, Keras, Liste 11-7'de gösterildiği gibi eğitim sırasında doğrulama verilerimizi basitçe fit\_generator () geçirmemize izin veriyor.

model.fit\_generator(

(1) validation\_data=validation\_data,

generator=training\_generator,

steps\_per\_epoch=steps\_per\_epoch,

epochs=10

)

Listing 11-7: Using validation data for automatic monitoring during training:Liste 11-7: Eğitim sırasında otomatik izleme için doğrulama verilerini kullanma

11-7 listesi, doğrulama\_verisinin artık fit\_generator (1) 'e geçirilmesi dışında, Liste 11-5'in sonuyla hemen hemen aynıdır. Bu, doğrulama kaybı ve doğruluğunun eğitim kaybı ve doğruluğu ile birlikte hesaplanmasını sağlayarak model izlemeyi geliştirmeye yardımcı olur.

Şimdi, eğitim ifadeleri Şekil 11-3'e benzemelidir.

Figure 11-3: Console output from training a Keras model with validation data:Şekil 11-3: Doğrulama verileriyle bir Keras modelinin eğitiminden elde edilen konsol çıktısı

Şekil 11-3, Şekil 11-2'ye benzer, tek fark, her dönem için yalnızca eğitim kaybını ve ak metriklerini acc metrics göstermek yerine, artık her dönem için val\_loss (doğrulama kaybı) ve val\_acc (doğrulama doğruluğu validation accuracy) da hesaplar ve gösterir. Genel olarak, doğrulama doğruluğu yukarı yerine aşağı gidiyorsa, bu, modelinizin eğitim verilerinize fazla uyduğunun bir göstergesidir ve eğitimi durdurmak en iyisidir. Doğrulama doğruluğu, burada olduğu gibi artıyorsa, bu, modelinizin hala daha iyi olduğu ve eğitime devam etmeniz gerektiği anlamına gelir.

\*Saving and Loading the Model:Modeli Kaydetme ve Yükleme

Artık bir sinir ağını nasıl oluşturacağınızı ve eğiteceğinizi bildiğinize göre, bunu başkalarıyla paylaşabilmek için nasıl kaydedeceğinizi gözden geçirelim.

11-8 listesi eğitimli modelimizi bir h5 dosyasına (1) nasıl kaydedeceğimizi ve onu (potansiyel olarak daha sonraki bir tarihte) yeniden yükleyeceğimizi (2) gösterir.

from keras.models import load\_model

# save the model

(1) model.save('my\_model.h5')

# load the model back into memory from the file:

(2)same\_model = load\_model('my\_model.h5')

Listing 11-8: Saving and loading Keras models

\*Evaluating the Model:Modeli Değerlendirme

Model eğitimi bölümünde, eğitim kaybı ve doğruluğu gibi bazı varsayılan model değerlendirme ölçütlerinin yanı sıra doğrulama kaybı ve doğruluğu gözlemledik. Modellerimizi daha iyi değerlendirmek için şimdi daha karmaşık metrikleri gözden geçirelim.

auc-> area under the curve

receive operating characteristic roc

İkili bir öngörücünün doğruluğunu değerlendirmek için kullanışlı bir metriğe eğri altındaki alan (AUC) denir. Eğri, tüm olası puan eşikleri için gerçek pozitif oranlara (y ekseni) karşı yanlış pozitif oranları (x ekseni) gösteren bir Alıcı Çalışma Karakteristiği (ROC) eğrisine (bkz. Bölüm 8) karşılık gelir.

Örneğin, modelimiz bir dosyanın kötü niyetli olup olmadığını 0 (iyi huylu) ile 1 (kötü niyetli) arasında bir puan kullanarak tahmin etmeye çalışır. Bir dosyayı kötü amaçlı olarak sınıflandırmak için nispeten yüksek bir puan eşiği seçersek, daha az sahte (iyi) ve aynı zamanda daha az gerçek pozitif (kötü) elde ederiz. Öte yandan, düşük bir puan eşiği seçersek, muhtemelen yüksek bir yanlış pozitif oranımız (kötü) ancak çok yüksek bir tespit oranımız (iyi) olacaktır.

Bu iki örnek olasılık, modelimizin ROC eğrisinde iki nokta olarak temsil edilecektir; burada birincisi eğrinin sol tarafına ve ikincisi sağ tarafa yakın konumlandırılacaktır. AUC, Şekil 11-4'te gösterildiği gibi, bu ROC eğrisinin altındaki alanı alarak tüm bu olasılıkları temsil eder.

Basit bir ifadeyle, 0,5'lik bir AUC, yazı tura atmanın tahmin kabiliyetini temsil ederken, 1'lik bir AUC mükemmeldir.

Şekil 11-4: Çeşitli örnek ROC eğrileri. Her bir ROC eğrisi (çizgi) farklı bir AUC değerine karşılık gelir.

Liste 11-9'daki kodu kullanarak doğrulama AUC'yi hesaplamak için doğrulama verilerimizi kullanalım.

from sklearn import metrics

(1) validation\_labels = validation\_data[1]

(2) validation\_scores = [el[0] for el in model.predict(validation\_data[0])]

(3) fpr, tpr, thres = metrics.roc\_curve(y\_true=validation\_labels,

y\_score=validation\_scores)

(4) auc = metrics.auc(fpr, tpr)

print('Validation AUC = {}'.format(auc))

Listing 11-9: Calculating validation AUC using sklearn’s metric submodule:Liste 11-9: sklearn’in metrik alt modülünü kullanarak doğrulama AUC'yi hesaplama

Burada, validation\_data demetimizi iki nesneye ayırıyoruz: validation\_labels (1) ile temsil edilen doğrulama etiketleri ve validation\_scores (2) ile temsil edilen düzleştirilmiş doğrulama modeli tahminleri. Ardından, model tahminleri için yanlış pozitif oranları, gerçek pozitif oranları ve ilişkili eşik değerlerini hesaplamak için sklearn'daki metrics.roc\_curve işlevini kullanırız (3). Bunları kullanarak, yine bir sklearn fonksiyonu (4) kullanarak AUC metriğimizi hesaplıyoruz.

Burada işlev kodunu gözden geçirmeyecek olsam da, gerçek ROC eğrisini Liste 11-10'da gösterildiği gibi bu kitapla birlikte verilen verilerdeki ch11 / model\_evaluation.py dosyasında bulunan roc\_plot () işlevini de kullanabilirsiniz. .

from ch11.model\_evaluation import roc\_plot

roc\_plot(fpr=fpr, tpr=tpr, path\_to\_file='roc\_curve.png')

Listing 11-10: Creating a ROC curve plot using the roc\_plot function from this book’s

accompanying data, in ch11/model\_evaluation.py:Liste 11-10: Bu kitabın eşlik eden verilerinden roc\_plot işlevini kullanarak, ch11 / model\_evaluation.py içinde bir ROC eğri grafiği oluşturma

Kodun Liste 11-10'da çalıştırılması, Şekil 11-5'e benzeyen bir çizim (roc\_curve.png'ye kaydedilmiş) oluşturmalıdır.

Şekil 11-5'teki ROC eğrisindeki her nokta, 0 ile 1 arasında değişen çeşitli model tahmin eşikleri ile ilişkili spesifik bir yanlış pozitif oranı (x ekseni) ve gerçek pozitif oranı (y ekseni) temsil eder. Yanlış pozitif oranlar arttıkça, doğru -pozitif oranlar artar ve bunun tersi de geçerlidir. Üretim ortamlarında, yanlış pozitifleri tolere etme isteğinize göre riske izin verme isteğinize bağlı olarak kararınızı vermek için genellikle tek bir eşik (bu eğri üzerinde tek bir nokta, doğrulama verilerinin üretim verilerini taklit ettiğini varsayarak) seçmeniz gerekir. kötü amaçlı dosya çatlaklardan sızmak için. a malicious file to slip through the cracks.

\*Enhancing the Model Training Process with Callbacks:Geri Çağırma ile Model Eğitim Sürecini Geliştirme

Şimdiye kadar, Keras modellerini nasıl tasarlayacağınızı, eğiteceğinizi, kaydedeceğinizi, yükleyeceğinizi ve değerlendireceğinizi öğrendiniz. Oldukça iyi bir başlangıç için ihtiyacınız olan tek şey bu olsa da, model eğitim sürecimizi daha da iyi hale getirebilecek Keras geri aramalarını da tanıtmak istiyorum.

Keras geri araması, eğitim sürecinin belirli aşamalarında Keras'ın uyguladığı bir dizi işlevi temsil eder. Örneğin, her dönemin sonunda bir h5 dosyasının kaydedildiğinden veya doğrulama AUC'nin her dönemin sonunda ekrana yazdırıldığından emin olmak için bir Keras geri araması kullanabilirsiniz. Bu, eğitim sürecinde modelinizin nasıl gittiğini daha kesin bir şekilde kaydetmenize ve size bilgi vermenize yardımcı olabilir.

Yerleşik bir geri arama kullanarak başlıyoruz ve ardından kendi özel geri aramamızı yazmaya çalışıyoruz.

\*Using a Built-in Callback:Yerleşik Geri Aramayı Kullanma

Yerleşik bir geri arama kullanmak için, eğitim sırasında modelinizin fit\_generator () yöntemine bir geri arama örneğini iletmeniz yeterlidir. Geri aramaları kullanacağız. Her eğitim döneminden sonra doğrulama kaybını değerlendiren ve doğrulama kaybı önceki dönemin doğrulama kayıplarından daha küçükse mevcut modeli bir dosyaya kaydeden ModelCheckpoint geri çağrısı. Bunu yapmak için, geri aramanın doğrulama verilerimize erişmesi gerekir, bu nedenle onu Liste 11-11'de gösterildiği gibi fit\_generator () yöntemine aktaracağız.

from keras import callbacks

model.fit\_generator(

generator=training\_generator,

# lowering steps\_per\_epoch so the example code runs fast:

steps\_per\_epoch=50,

epochs=5,

validation\_data=validation\_data,

callbacks=[

callbacks.ModelCheckpoint(save\_best\_only=True,(1)

(2) filepath='results/best\_model.h5',

(3)monitor='val\_loss')

],

)

Listing 11-11: Adding a ModelCheckpoint callback to the training process:Liste 11-11: Eğitim sürecine ModelCheckpoint geri araması ekleme

Bu kod, 'val\_loss' (3) (doğrulama kaybı) yeni bir düşük seviyeye ulaştığında, modelin (1) tek bir dosyaya 'results / best\_model.h5' (2) üzerine yazılmasını sağlar. Bu, geçerli kaydedilen modelin ('results / best\_model .h5') her zaman doğrulama kaybıyla ilgili olarak tamamlanan tüm çağlarda en iyi modeli temsil etmesini sağlar.

Alternatif olarak, modeli her dönemden sonra doğrulama kaybından bağımsız olarak ayrı bir dosyaya kaydetmek için Liste 11-12'deki kodu kullanabiliriz.

callbacks.ModelCheckpoint(save\_best\_only=False,(4)

                                                        (5) filepath='results/model\_epoch\_{epoch}.h5',

monitor='val\_loss')

Listing 11-12: Adding a ModelCheckpoint callback to the training process that saves the

model to a different file after each epoch:Liste 11-12: Eğitim sürecine her dönemden sonra modeli farklı bir dosyaya kaydeden bir ModelCheckpoint geri araması ekleme

Bunu yapmak için, Liste 11-11'de aynı kodu ve aynı ModelCheckpoint işlevini kullanıyoruz, ancak save\_best\_only = False (4) ve Keras'tan epoch numarasını (5) doldurmasını isteyen bir dosya yolu kullanıyoruz. Modelimizin yalnızca "en iyi" sürümünü kaydetmek yerine, Liste 11-12’nin geri araması, her çağın modelimizin sürümünü sonuç / model\_epoch\_0.h5, sonuçlar / model\_epoch\_1.h5, sonuçlar / model\_epoch\_2.h5 vb. Olarak kaydeder.

\*Using a Custom Callback:Özel Geri Aramayı Kullanma

Keras, AUC'yi desteklemese de, örneğin, her dönemden sonra ekrana AUC yazdırmamıza izin vermek için kendi özel geri aramamızı tasarlayabiliriz.

Özel bir Keras geri araması oluşturmak için, yeni geri aramalar oluşturmak için kullanılan soyut temel sınıf olan keras.callbacks.Callback'ten miras alan bir sınıf oluşturmamız gerekir. Adlarının belirttiği zamanlarda, eğitim sırasında otomatik olarak çalıştırılacak bir veya daha fazla yöntem seçimi ekleyebiliriz: on\_epoch\_begin, on\_epoch\_end, on\_batch\_begin, on\_batch\_end, on\_train\_begin ve on\_train\_end. Liste 11-13, doğrulama AUC'sini hesaplayan ve her dönemin sonunda ekrana yazdıran bir geri aramanın nasıl oluşturulacağını gösterir.

import numpy as np

from keras import callbacks

from sklearn import metrics

(1) class MyCallback(callbacks.Callback):

(2)def on\_epoch\_end(self, epoch, logs={}):

(3) validation\_labels = self.validation\_data[1]

validation\_scores = self.model.predict(self.validation\_data[0])

# flatten the scores:

validation\_scores = [el[0] for el in validation\_scores]

fpr, tpr, thres = metrics.roc\_curve(y\_true=validation\_labels,

            y\_score=validation\_scores)

(4) auc = metrics.auc(fpr, tpr)

print('\n\tEpoch {}, Validation AUC = {}'.format(epoch,

                                        np.round(auc, 6)))

model.fit\_generator(

generator=training\_generator,

# lowering steps\_per\_epoch so the example code runs fast:

steps\_per\_epoch=50,

epochs=5,

(5) validation\_data=validation\_data,

(6)callbacks=[

callbacks.ModelCheckpoint('results/model\_epoch\_{epoch}.h5',

            monitor='val\_loss',

            save\_best\_only=False,

            save\_weights\_only=False)

]

)

Listing 11-13: Creating and using a custom callback to print AUC to the screen after each

training epoch:Liste 11-13: Her eğitim döneminden sonra AUC'yi ekrana yazdırmak için özel bir geri arama oluşturma ve kullanma

    Bu örnekte, önce geri aramalardan miras alan MyCallback sınıfımızı (1) oluşturuyoruz. İşleri basit tutarak, tek bir yöntemin üzerine yazıyoruz, on\_epoch\_end (2) ve ona Keras tarafından beklenen iki argüman veriyoruz: epoch ve logs (bir günlük bilgileri sözlüğü), her ikisi de Keras eğitim sırasında fonksiyonu çağırdığında sağlayacaktır.

  Ardından, geri çağırmalar sayesinde öz nesnede zaten depolanan validation\_data (3) 'yı alırız. Geri arama mirası ve "Modeli Değerlendirme" sayfa 209'da yaptığımız gibi AUC'yi (4) hesaplar ve yazdırırız. bu kodun çalışması için, geri aramanın eğitim sırasında self.validation\_data erişimine sahip olması için doğrulama verilerinin fit\_generator () 'a geçirilmesi gerekir (5). Son olarak, modele yeni geri aramamızı eğitmesini ve belirtmesini söyleriz (6). Sonuç Şekil 11-6 gibi görünmelidir.

Figure 11-6: Console output from training a Keras model with a custom AUC callback:Şekil 11-6: Özel bir AUC geri aramasıyla bir Keras modelinin eğitiminden elde edilen konsol çıktısı

Gerçekten önemsediğiniz şey doğrulama AUC'sini en aza indirmekse, bu geri arama, modelinizin eğitim sırasında nasıl çalıştığını görmenizi kolaylaştırarak eğitim sürecini durdurmanız gerekip gerekmediğini değerlendirmenize yardımcı olur (örneğin, doğrulama doğruluğu zaman içinde sürekli olarak azalırsa) ).

->ÖZET:

Bu bölümde, Keras kullanarak kendi sinir ağınızı nasıl kuracağınızı öğrendiniz. Ayrıca onu eğitmeyi, değerlendirmeyi, kaydetmeyi ve yüklemeyi de öğrendiniz. Daha sonra, yerleşik ve özel geri aramalar ekleyerek model eğitim sürecini nasıl geliştireceğinizi öğrendiniz. Model mimarisi ve özellik çıkarımının model doğruluğu üzerinde ne gibi değişikliklere sahip olabileceğini görmek için bu kitapla birlikte verilen kodla oynamanızı tavsiye ederim.

Bu bölüm ayaklarınızı ıslatmak içindir, ancak bir başvuru kılavuzu olarak tasarlanmamıştır. En güncel resmi belgeler için https://keras.io adresini ziyaret edin. Keras'ın ilginizi çeken yönlerini araştırmak için zaman ayırmanızı şiddetle tavsiye ediyorum. Umarım bu bölüm, tüm güvenlik derin öğrenme maceralarınız için iyi bir başlangıç noktası olarak hizmet etmiştir!