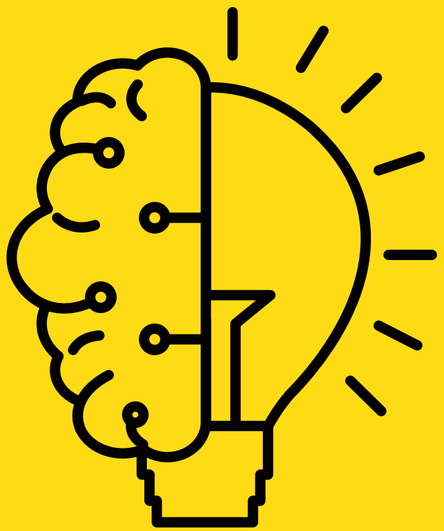


DERİN ÖĞRENMEYE GİRİŞ

Generative Adversarial Networks

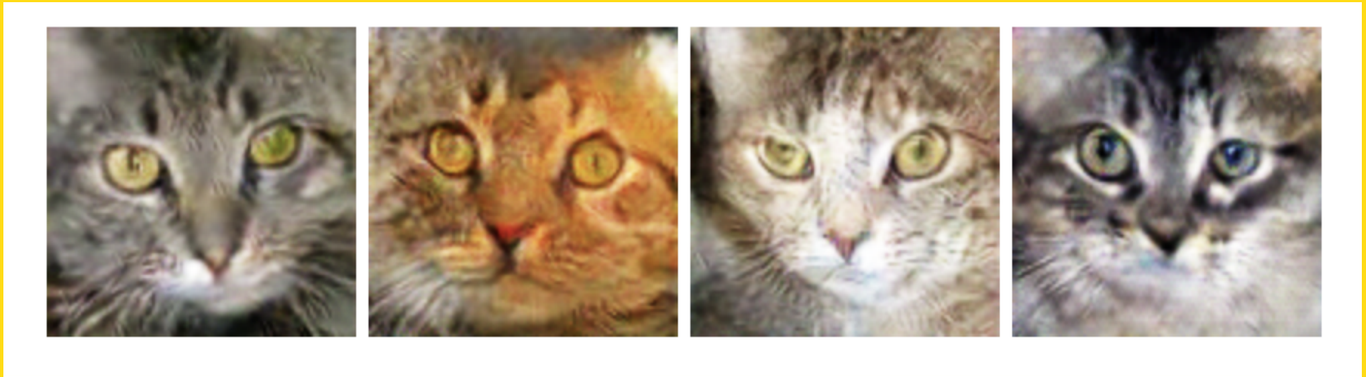
Gülşah Sevinel b171200025

GENERATIVE ADVERSARIAL NETWORKS

ÇEKİŞMELİ ÜRETİCİ AĞLAR

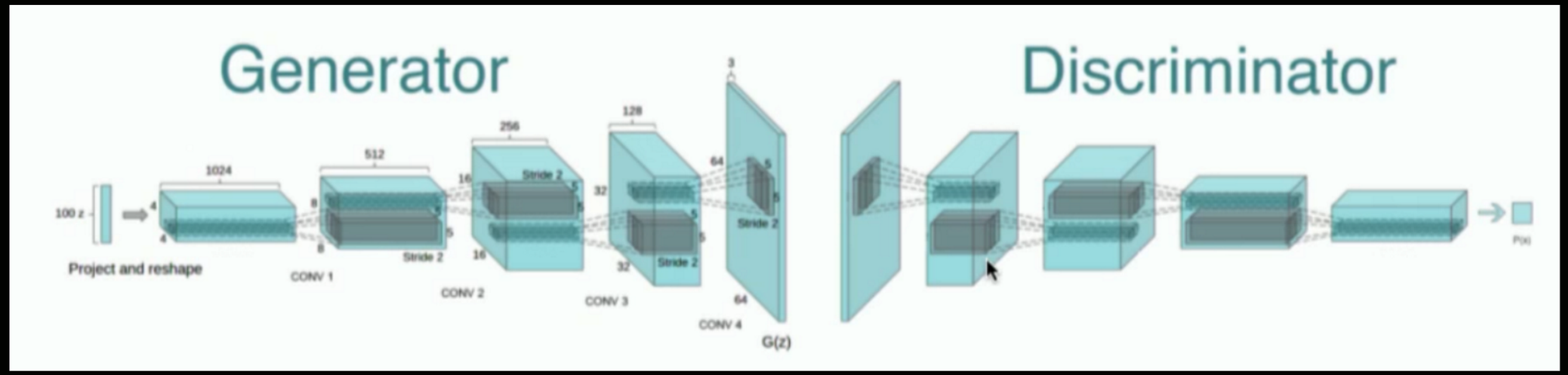
Nerdeyse gerçekle farkı olmayan yeni resimler üreten bir yapay sinir ağları modeli.
2014 yılında Ian Goodfellow ve çalışma arkadaşları tarafından NIPS konferansında tanıtılmıştır.

Generated cats



Generated cats - model 2

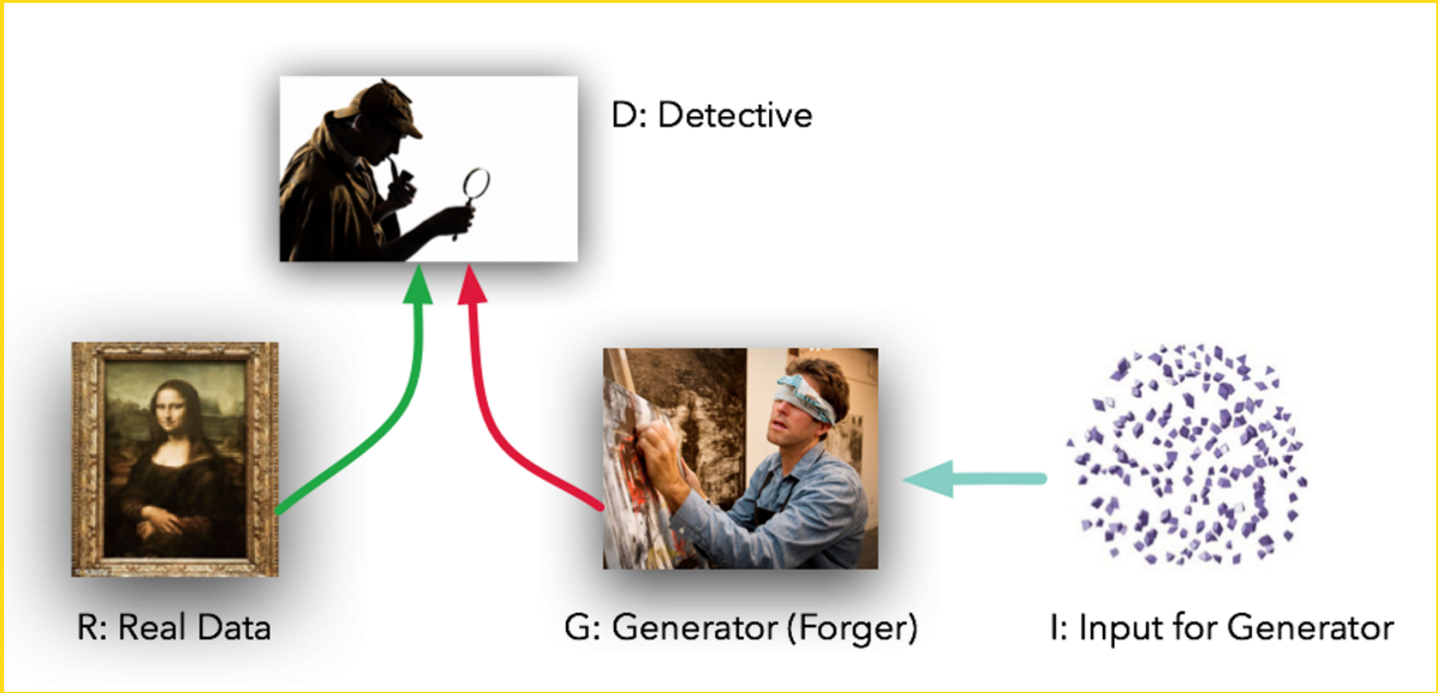
Bu modele yeteri kadar veri verirseniz, verdiğiniz veriden yeni örnekler üretebilirsiniz. Örneğin sisteme binlerce kedi fotoğrafı verirseniz sistem bir kedinin nasıl görünmesi gerektiğini öğrenecek ve size daha önce var olmamış kedi fotoğrafları üretecektir.



GAN (Çekişmeli Üretici Ağlar) iki farklı ağ yağısından oluşur.

- Generator (Üretici)
- Discriminator (Ayırt Edici)

Generator, herhangi bir girdiden gerçeğe benzeyen yeni resimler,sesler, modeller vb. üretir. Discriminator ise sahte ve gerçek verileri birbirinden ayırt etmeye çalışır.



Bu ikiliyi bir örnek olarak sahtekar (generator) ve dedektif (discriminator) ikilisine benzetebiliriz.

Ünlü ressamların adını kullanarak gerçek tablolara benzer tablolar üreten sahtekar, bu resimlerin gerçek olup olmadığına karar veren ise dedektiftir. Dedektif sahtekarın yaptığı tabloların gerçek olmadığını anladıkça sahtekar daha gerekçi tablolar yapmaya başlayacaktır.

BU SÜREÇ NASIL İŞLİYOR?

HER İKİ AĞDA AYNI ANDA EĞİTİLİR.

"Eğitimin her döneminde küçük toptanlar (minibach) halinde veri seti örnekleri alınıyor ve her bir dönemde (epoch) ayırt edici ağ üzerinde artan rastgele dereceli azaltma (stochastic gradient descent) uygulanırken üretici ağ üzerinde azalan rastgele dereceli azalma uygulanıyor. Bu şekilde bu iki ağın geri yayılım algoritması ile ağırlıkları eğitiliyor ve sonunda bir dengeye ulaşmaları matematiksel olarak sağlanıyor. Bu dengede artık optimum kabul edilebilir yeni veriler üretilmeye ve bu üretilen veriler optimum öğrenmiş ayırt edici ağ tarafından kabul edilmeye başlıyor."

Bu, TensorFlow ile oluşturulmuş çok basit, üretken bir rakip ağın bir örneğidir. MNIST veri kümesinden el yazısıyla yazılmış rakamlara benzeyen görüntüler üretir.

```
[ ]
```

```python
import tensorflow as tf #makine öğrenimi
import numpy as np #matrix math
import datetime #model kontrol noktaları ve eğitim için zamanı kaydetme
import matplotlib.pyplot as plt #sonuçları görselleştirme için
%matplotlib inline

# 1. Adım - Veri kümesi toplama
#MNIST -  çeşitli görüntü işleme sistemlerini eğitmek için yaygın olarak kullanılan büyük bir el yazısı rakam veritabanıdır.
from tensorflow.examples.tutorials.mnist import input_data
#doğru verilerin bilgisayarınıza indirildiğinden emin olur
#local eğitim klasörüne gidin ve ardından DataSet örneklerinin sözlüğünü döndürmek için bu verileri paketinden çıkarın.
mnist = input_data.read_data_sets("MNIST_data/")
```

```python
def discriminator(x_image, reuse=False):
    if (reuse):
        tf.get_variable_scope().reuse_variables()
    # İlk evrişimli ve havuz katmanları
    # Bunlar 32 farklı 5 x 5 piksel özelliği arıyor
    # Görüntüyü evrişimli bir katmandan geçirerek başlayacağız.
    # İlk olarak, tf.get_variable aracılığıyla ağırlık ve önyargı değişkenlerimizi oluşturuyoruz.
    # İlk ağırlık matrisimiz (veya filtremiz) 5x5 boyutunda olacak ve çıktı derinliği 32 olacaktır.
    # Normal bir dağılımdan rastgele başlatılacaktır.
    d_w1 = tf.get_variable('d_w1', [5, 5, 1, 32], initializer=tf.truncated_normal_initializer(stddev=0.02))
    #tf.constant_init sabit değerli tensörler üretir.
    d_b1 = tf.get_variable('d_b1', [32], initializer=tf.constant_initializer(0))
    #tf.nn.conv2d() ortak bir evrişim için Tensorflow'un işlevidir.
    # 4 argüman alır. İlki giriş hacmi (bu durumda 28 x 28 x 1 resmimiz).
    # Sonraki argüman filtre / ağırlık matrisidir. Son olarak, adımınızı değiştirebilir ve
    # evrişimin tamponlanması. Bu iki değer çıktı hacminin boyutlarını etkiler.
    # "AYNI" sola ve sağa eşit olarak doldurmaya çalışır, ancak eklenecek sütun sayısı tuhafsa,
    # ekstra sütunu sağa ekleyecektir,
    #strides = [toplu iş, yükseklik, genişlik, kanallar]
    d1 = tf.nn.conv2d(input=x_image, filter=d_w1, strides=[1, 1, 1, 1], padding='SAME')
    #bias'ı ekle
    d1 = d1 + d_b1
    #ReLU
    d1 = tf.nn.relu(d1)
    # Ortalama bir havuzlama katmanı, girdiyi bölerek alt örnekleme gerçekleştirir.
    # Dikdörtgen havuzlama bölgeleri ve her bölgenin ortalamasını hesaplama.
    # Havuz bölgeleri için ortalamaları döndürür.
    d1 = tf.nn.avg_pool(d1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
```

```
# Herhangi bir evrişimli sinir ağında olduğu gibi, bu modül tekrarlanır,
# İkinci evrişimli ve havuz katmanları
# 64 farklı 5 x 5 piksel özelliği arar
d_w2 = tf.get_variable('d_w2', [5, 5, 32, 64], initializer=tf.truncated_normal_initializer(stddev=0.02))
d_b2 = tf.get_variable('d_b2', [64], initializer=tf.constant_initializer(0))
d2 = tf.nn.conv2d(input=d1, filter=d_w2, strides=[1, 1, 1, 1], padding='SAME')
d2 = d2 + d_b2
d2 = tf.nn.relu(d2)
d2 = tf.nn.avg_pool(d2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

# ve ardından bir dizi tam bağlantılı katman.
# İlk tamamen bağlı katman
d_w3 = tf.get_variable('d_w3', [7 * 7 * 64, 1024], initializer=tf.truncated_normal_initializer(stddev=0.02))
d_b3 = tf.get_variable('d_b3', [1024], initializer=tf.constant_initializer(0))
d3 = tf.reshape(d2, [-1, 7 * 7 * 64])
d3 = tf.matmul(d3, d_w3)
d3 = d3 + d_b3
d3 = tf.nn.relu(d3)

# Tamamen bağlantılı son katman, sınıf puanları gibi çıktıları tutar.
 # İkinci tam bağlantılı katman
d_w4 = tf.get_variable('d_w4', [1024, 1], initializer=tf.truncated_normal_initializer(stddev=0.02))
d_b4 = tf.get_variable('d_b4', [1], initializer=tf.constant_initializer(0))

# Ağın sonunda, son bir matris çarpımı yaparız ve
 # aktivasyon değerini geri döndürür.
 # CNN'lerde rahat olanlar için, bu sadece basit bir ikili sınıflandırıcıdır.
 # Son katman
d4 = tf.matmul(d3, d_w4) + d_b4
# d4 boyutları: batch_size x 1
return d4
```

```python
#You can think of the generator as being a kind of reverse ConvNet. With CNNs, the goal is to
#transform a 2 or 3 dimensional matrix of pixel values into a single probability. A generator,
#however, seeks to take a d-dimensional noise vector and upsample it to become a 28 x 28 image.
#ReLUs are then used to stabilize the outputs of each layer.

#it rastgele girdiler alır ve sonunda bunları MNIST veri şekline uyması için [1,28,28] piksele eşler.
# Yoğun bir 14 × 14 değer kümesi oluşturarak başlayın ve ardından bir avuç dolusu filtreden geçirin.
# değişken boyutlar ve kanal sayısı
# ağırlık matrisleri giderek küçülüyor

def generator(batch_size, z_dim):
    z = tf.truncated_normal([batch_size, z_dim], mean=0, stddev=1, name='z')
    #ilk deconv bloğu
    g_w1 = tf.get_variable('g_w1', [z_dim, 3136], dtype=tf.float32, initializer=tf.truncated_normal_initializer(stddev=0.02))
    g_b1 = tf.get_variable('g_b1', [3136], initializer=tf.truncated_normal_initializer(stddev=0.02))
    g1 = tf.matmul(z, g_w1) + g_b1
    g1 = tf.reshape(g1, [-1, 56, 56, 1])
    g1 = tf.contrib.layers.batch_norm(g1, epsilon=1e-5, scope='bn1')
    g1 = tf.nn.relu(g1)

    # 50 feature oluşturur
    g_w2 = tf.get_variable('g_w2', [3, 3, 1, z_dim/2], dtype=tf.float32, initializer=tf.truncated_normal_initializer(stddev=0.02))
    g_b2 = tf.get_variable('g_b2', [z_dim/2], initializer=tf.truncated_normal_initializer(stddev=0.02))
    g2 = tf.nn.conv2d(g1, g_w2, strides=[1, 2, 2, 1], padding='SAME')
    g2 = g2 + g_b2
    g2 = tf.contrib.layers.batch_norm(g2, epsilon=1e-5, scope='bn2')
    g2 = tf.nn.relu(g2)
    g2 = tf.image.resize_images(g2, [56, 56])
```

```python
# 25 feature oluşturur
g_w3 = tf.get_variable('g_w3', [3, 3, z_dim/2, z_dim/4], dtype=tf.float32, initializer=tf.truncated_normal_initializer(stddev=0.02))
g_b3 = tf.get_variable('g_b3', [z_dim/4], initializer=tf.truncated_normal_initializer(stddev=0.02))
g3 = tf.nn.conv2d(g2, g_w3, strides=[1, 2, 2, 1], padding='SAME')
g3 = g3 + g_b3
g3 = tf.contrib.layers.batch_norm(g3, epsilon=1e-5, scope='bn3')
g3 = tf.nn.relu(g3)
g3 = tf.image.resize_images(g3, [56, 56])

# Tek çıkış kanallı son evrişim
g_w4 = tf.get_variable('g_w4', [1, 1, z_dim/4, 1], dtype=tf.float32, initializer=tf.truncated_normal_initializer(stddev=0.02))
g_b4 = tf.get_variable('g_b4', [1], initializer=tf.truncated_normal_initializer(stddev=0.02))
g4 = tf.nn.conv2d(g3, g_w4, strides=[1, 2, 2, 1], padding='SAME')
g4 = g4 + g_b4
g4 = tf.sigmoid(g4)

# Son katmanda toplu normalleştirme yok, ancak ekliyoruz
# Üretilen görüntüleri daha net hale getirmek için bir sigmoid aktivatör.
# G4 boyutları: batch_size x 28 x 28 x 1

return g4
```

```
sess = tf.Session()

batch_size = 50
z_dimensions = 100

x_placeholder = tf.placeholder("float", shape = [None,28,28,1], name='x_placeholder')
# x_placeholder girdi görüntülerini ayırıcıya beslemek için



# Generator, daha gerçekçi görüntüler üretmek için sürekli olarak gelişirken, ayırt edici
# gerçek ve oluşturulmuş görüntüleri ayırt etmede daha iyi ve daha iyi olmaya çalışır.
# Bu, her iki ağı da etkileyen kayıp işlevlerini formüle etmemiz gerektiği anlamına gelir.

Gz = generator(batch_size, z_dimensions)
# Gz oluşturulan görüntüleri tutar
#g(z)

Dx = discriminator(x_placeholder)
# Dx gerçek MNIST görüntüleri için ayrımcıları tahmin olasılıklarını tutar
#d(x)

Dg = discriminator(Gz, reuse=True)
# Dg, oluşturulan görüntüler için ayırt edici tahmin olasılıklarını tutar
#d(g(z))
```

```
# Önce Generator ağının yaratılmasını istiyoruz
# Ayrımcıyı yanıltacak görüntüler. Generator, ayrımcının bir 1 (pozitif örnek) vermesini ister.
# Bu nedenle, Dg ile 1'in etiketi arasındaki kaybı hesaplamak istiyoruz. Bu,
# tf.nn.sigmoid_cross_entropy_with_logits işlevi. Bu, çapraz entropi kaybının
# iki argüman arasında alınmalıdır. "With_logits" bileşeni, işlevin ölçeklenmemiş değerlerde çalışacağı anlamına gelir
# Temel olarak bu, çıktıyı sıkıştırmak için bir softmax işlevi kullanmak yerine
# activations 0'dan 1'e olasılık değerlerine, basitçe matris çarpımının ölçeklenmemiş değerini döndürüyoruz.
# Ayrımcımızın son satırına bir göz atın. Sonunda softmax veya sigmoid katman yok.
# Ortalama azaltma işlevi, döndürülen matrixx'teki tüm bileşenlerin ortalama değerini çapraz entropi işlevi ile alır.
# Bu, kaybı bir vektör veya matris yerine tek bir skaler değere düşürmenin bir yoludur.


g_loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=Dg, labels=tf.ones_like(Dg)))


# Şimdi, ayrımcının bakış açısını düşünelim. Amacı sadece doğru etiketleri elde etmektir
# (her MNIST basamağı için çıkış 1 ve oluşturulanlar için 0). Dx arasındaki kaybı hesaplamak istiyoruz
# ve 1'in doğru etiketinin yanı sıra Dg ile 0 arasındaki kaybı düzeltmek istiyoruz.
d_loss_real = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=Dx, labels=tf.fill([batch_size, 1], 0.9)))
d_loss_fake = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=Dg, labels=tf.zeros_like(Dg)))
d_loss = d_loss_real + d_loss_fake

tvars = tf.trainable_variables()

d_vars = [var for var in tvars if 'd_' in var.name]
g_vars = [var for var in tvars if 'g_' in var.name]
```

```
# Ayrımcıyı eğitimi
# GitHub sürümünde 0,001'den artış
with tf.variable_scope(tf.get_variable_scope(), reuse=False) as scope:
    # Sonra, iki optimize edicimizi belirteceğiz. Bugünün derin öğrenme çağında Adam,
    # Uyarlanabilir öğrenme hızları ve momentumu kullandığı için en iyi SGD optimize edici.
    # Adam'ın simge durumuna küçültme işlevini çağırıyoruz ve ayrıca güncellemesini istediğimiz değişkenleri de belirtiyoruz.
    d_trainer_fake = tf.train.AdamOptimizer(0.0001).minimize(d_loss_fake, var_list=d_vars)
    d_trainer_real = tf.train.AdamOptimizer(0.0001).minimize(d_loss_real, var_list=d_vars)

    # generator eğitimi
    # GitHub sürümünde 0.004'ten düşüyor
    g_trainer = tf.train.AdamOptimizer(0.0001).minimize(g_loss, var_list=g_vars)
```

```
# Tek bir skaler değer içeren bir özet protokol değerini çıkarır.
tf.summary.scalar('Generator_loss', g_loss)
tf.summary.scalar('Discriminator_loss_real', d_loss_real)
tf.summary.scalar('Discriminator_loss_fake', d_loss_fake)

d_real_count_ph = tf.placeholder(tf.float32)
d_fake_count_ph = tf.placeholder(tf.float32)
g_count_ph = tf.placeholder(tf.float32)

tf.summary.scalar('d_real_count', d_real_count_ph)
tf.summary.scalar('d_fake_count', d_fake_count_ph)
tf.summary.scalar('g_count', g_count_ph)

# Ayrımcının nasıl değerlendirdiğini görmek için sağlık kontrolü
# oluşturulmuş ve gerçek MNIST görüntüleri
d_on_generated = tf.reduce_mean(discriminator(generator(batch_size, z_dimensions)))
d_on_real = tf.reduce_mean(discriminator(x_placeholder))

tf.summary.scalar('d_on_generated_eval', d_on_generated)
tf.summary.scalar('d_on_real_eval', d_on_real)

images_for_tensorboard = generator(batch_size, z_dimensions)
tf.summary.image('Generated_images', images_for_tensorboard, 10)
merged = tf.summary.merge_all()
logdir = "tensorboard/gan/"
writer = tf.summary.FileWriter(logdir, sess.graph)
print(logdir)
```

```python
saver = tf.train.Saver()

sess.run(tf.global_variables_initializer())

# Her yineleme sırasında, biri ayırıcıya diğeri de oluşturucuya olmak üzere iki güncelleme yapılacaktır.
# Oluşturucu güncellemesi için, oluşturucuya rastgele bir z vektörü besleyeceğiz ve bu çıktıyı
# olasılık puanı elde etmek için ayırıcıya ileteceğiz (bu, daha önce belirttiğimiz Dg değişkenidir).
# Kayıp fonksiyonumuzdan hatırladığımız gibi, çapraz entropi kaybı en aza indirilir,
# ve yalnızca generator'ın ağırlıkları ve önyargıları güncellenir.
# Ayırıcı güncellemesi için de aynısını yapacağız.
# Programımızın başında yarattığımız mnist değişkeninden bir grup resim alacağız.
# Bunlar olumlu örnekler sunarken, önceki bölümdeki görseller olumsuz olanlardır.

gLoss = 0
dLossFake, dLossReal = 1, 1
d_real_count, d_fake_count, g_count = 0, 0, 0
for i in range(50000):
    real_image_batch = mnist.train.next_batch(batch_size)[0].reshape([batch_size, 28, 28, 1])
    if dLossFake > 0.6:
        # Ayrıştırıcıyı oluşturulan görüntüler üzerinde eğitin
        _, dLossReal, dLossFake, gLoss = sess.run([d_trainer_fake, d_loss_real, d_loss_fake, g_loss],
                                                  {x_placeholder: real_image_batch})
        d_fake_count += 1

    if gLoss > 0.5:
        # generator eğitimi
        _, dLossReal, dLossFake, gLoss = sess.run([g_trainer, d_loss_real, d_loss_fake, g_loss],
                                                  {x_placeholder: real_image_batch})
        g_count += 1
```

```
if dLossReal > 0.45:
    # Ayrımcı gerçek görüntüleri sahte olarak sınıflandırırsa,
    # Ayrımcıyı gerçek değerler konusunda eğitme
    _, dLossReal, dLossFake, gLoss = sess.run([d_trainer_real, d_loss_real, d_loss_fake, g_loss],
                                              {x_placeholder: real_image_batch})
    d_real_count += 1

if i % 10 == 0:
    real_image_batch = mnist.validation.next_batch(batch_size)[0].reshape([batch_size, 28, 28, 1])
    summary = sess.run(merged, {x_placeholder: real_image_batch, d_real_count_ph: d_real_count,
                                d_fake_count_ph: d_fake_count, g_count_ph: g_count})
    writer.add_summary(summary, i)
    d_real_count, d_fake_count, g_count = 0, 0, 0

if i % 1000 == 0:
    # Defterde düzenli olarak örnek bir görüntü görüntüleyin
    # (Bunlar ayrıca her 10 yinelemede bir TensorBoard'a gönderilir)
    images = sess.run(generator(3, z_dimensions))
    d_result = sess.run(discriminator(x_placeholder), {x_placeholder: images})
    print("TRAINING STEP", i, "AT", datetime.datetime.now())
    for j in range(3):
        print("Discriminator classification", d_result[j])
        im = images[j, :, :, 0]
        plt.imshow(im.reshape([28, 28]), cmap='Greys')
        plt.show()

if i % 5000 == 0:
    save_path = saver.save(sess, "models/pretrained_gan.ckpt", global_step=i)
    print("saved to %s" % save_path)
```

```
est_images = sess.run(generator(10, 100))
test_eval = sess.run(discriminator(x_placeholder), {x_placeholder: test_images})

real_images = mnist.validation.next_batch(10)[0].reshape([10, 28, 28, 1])
real_eval = sess.run(discriminator(x_placeholder), {x_placeholder: real_images})

# Oluşturulan görüntüler için ayrımcı olasılıkları gösterin,
# ve görüntüleri görüntüleyin
for i in range(10):
    print(test_eval[i])
    plt.imshow(test_images[i, :, :, 0], cmap='Greys')
    plt.show()

# Şimdi aynısını gerçek MNIST görüntüleri için yapın
for i in range(10):
    print(real_eval[i])
    plt.imshow(real_images[i, :, :, 0], cmap='Greys')
    plt.show()
```