GoogLeNet

https://youtu.be/C86ZXvgpejM?si=HTMwte46uszlpIOI

Özellik	VGG16	GoogLeNet
Derinlik	16 katman	22 katman
Parametre sayısı	~138M	~5M
Yapı	Sıralı	Paralel (Inception)
Hız	Yavaş	Hızlı
Hafıza ihtiyacı	Yüksek	Düşük

🔍 GoogLeNet (Inception v1) Mimarisine Genel Bakış

Yıl: 2014
Google tarafından geliştirildi.
Başarı: ILSVRC 2014'te sınıflandırma yarışmasını kazandı.

□ Temel Fikir: Inception Bloğu

GoogLeNet'in kalbi **Inception bloğudur.**
Bu blok, klasik CNN mantığından farklı olarak **birden fazla filtre boyutunu aynı anda kullanır.**

Inception bloğu şu yolları paralel çalıştırır:

- **1x1 Conv**
- **1x1 Conv → 3x3 Conv**
- **1x1 Conv → 5x5 Conv**
- **3x3 MaxPooling → 1x1 Conv**

Sonra bu dört yolun çıktısı birleştirilir (concatenate).
Bu sayede model hem:

- küçük detayları (1x1, 3x3)
- hem de daha geniş bağlamları (5x5) öğrenebilir.

🎯 Neden Böyle Bir Yapı?

- Klasik CNN'lerde kernel boyutunu sabit seçmek zorundasın.
- Ama neyin daha iyi çalışacağını bilmiyoruz.
- Inception, **hepsini paralel çalıştırarak** modelin bunu öğrenmesine izin verir.
- Aynı zamanda **1x1 Conv** ile ara katmanlarda boyut azaltma yapılır. Bu sayede:
    - **Parametre sayısı düşer**
    - **Hız artar**
    - **Overfitting riski azalır**

🏗️ GoogLeNet Mimarisi Katmanları (Genel Yapı)

Aşama	Yapı
Giriş	224x224x3 görüntü
İlk katmanlar	7x7 Conv + MaxPooling + 1x1 + 3x3 Conv blokları
Orta katmanlar	9 adet Inception bloğu (3 blok grubu)
Yardımcı çıkışlar	2 tane Auxiliary Classifier (sadece eğitimde kullanılır)
Son katmanlar	Average Pooling → Dropout → Dense(1000) (ImageNet için)

□ Özellikleri

- **Derin ama verimli:** Yaklaşık 22 katman ama sadece ~5 milyon parametre (VGG16'ya göre çok az).
- **Paralel yapı:** Genişliği artırır, farklı bilgi seviyelerini aynı anda öğrenir.
- **Auxiliary classifier'lar:** Eğitim sırasında derin katmanlara gradyan ulaşmasını kolaylaştırmak için eklenir.

Uygulaman İçin Ne Anlama Geliyor?

Senin görevin:

- 224x224'lik görselleri sınıflandırmak (dolu/boş).
- Inception blokları sayesinde:
    - Yol dokusu, gölge, araç parçası gibi **farklı ölçekli detayları** öğrenebilir.
- Parametre sayısı düşük olduğu için **verimli çalışır**.
- Transfer learning ile kullandığında az sayıda veriyle bile **başarılı sonuçlar** alabilirsin.

Target_size nedir?

`target_size` parametresi, görüntüleri modele verirken **hangi boyuta yeniden boyutlandırmak istediğimizi** belirler.

📐 `target_size=(224, 224)` Ne Anlama Geliyor?

- Bu parametre, `ImageDataGenerator.flow_from_directory()` fonksiyonunda kullanılır.
- Veri setindeki tüm görselleri otomatik olarak **224 piksel genişlik ve 224 piksel yükseklik** olacak şekilde **yeniden boyutlandırır**.

```
python
Kodu kopyala
train_generator = train_datagen.flow_from_directory(
    "dataset/train",
    target_size=(224, 224),  # <--- işte burası
    ...
)
```

🎯 Neden 224x224?

- GoogLeNet (Inception v1) ve birçok klasik CNN mimarisi (VGG, ResNet) için **standart giriş boyutu** 224x224'tür.
- Model bu boyuttaki tensörleri bekler. Eğer daha büyük ya da küçük veriler verirsen, `Input shape mismatch` hatası alırsın.

🖼️ Örnek:

Varsayalım veri setinde 300x300 ve 512x512 gibi farklı boyutlarda resimler var.
Sen `target_size=(224,224)` dersen, bu görsellerin tümü modele girmeden önce şu şekilde yeniden boyutlandırılır:

```
scss
Kodu kopyala
(300x300) → (224x224)
(512x512) → (224x224)
```

Bu sayede **model sabit boyutlu tensorlerle eğitilir** ve GPU verimli çalışır.

💡 Not:

- UNet gibi bazı modeller için **kare olmayan** girişler de kullanılabilir (örneğin `target_size=(256,128)`).
- Ancak GoogLeNet ve benzeri klasik mimariler için genellikle kare ve sabit (`224x224`) giriş beklenir.

Her zaman train generatorda ne yapıyorsak validation ve test generatorda da mı aynısını yapmamız gerekiyor?

🎯 Kısa Cevap:

Hayır, her şeyi birebir aynı yapmak zorunda değilsin.
Ama bazı şeyleri kesinlikle aynı, bazılarını kesinlikle farklı yapmalısın.

🔍 Detaylı Açıklama:

1. Olmazsa Olmaz – Aynı Olmalı:

Özellik	***Neden Aynı Olmalı?***
target_size	Model sabit boyut bekler. Girdi boyutları eşleşmezse hata verir.
rescale	Eğitimde ve testte piksel değerleri aynı ölçekte olmalı. (0–255 → 0–1)
class_mode	Tüm setlerde etiketleme tutarlı olmalı (örneğin 'binary', 'categorical')

2. Olmamalı –Farklı Olmalı:

Özellik	***Neden Farklı Olmalı?***
augmentation (horizontal_flip, zoom, rotation)	Sadece eğitim setinde yapılmalı. Validation ve test verisi gerçek veriyi temsil etmeli, yapay olarak bozulmamalı.

3. shuffle Parametresi

- **Train**: `shuffle=True` olmalı → model her epoch'ta farklı örnek sıralaması görsün.
- **Test/Val**: `shuffle=False` olmalı → metrikler ve görsel çıktılar için sıralama tutarlı kalsın.

🧿 1. Shuffle Mekanizması Nasıl Çalışır?

`shuffle=True` yaptığında, `ImageDataGenerator` ya da `tf.data.Dataset` gibi veri akış araçları, her **epoch başında verilerin sırasını karıştırır.**

Örnek:

Diyelim ki elimizde 1000 örnek var: `[0, 1, 2, ..., 999]`

- **1. epoch:** `[33, 582, 7, 91, 10, ...]`
- **2. epoch:** `[912, 121, 3, 405, 2, ...]`
- Yani her epoch'ta model **farklı sırada veri görür.**

Neden önemli?

- Modelin **sıralamaya bağlı kalmadan** genelleme yapmasını sağlar.
- Aynı veri ardışık şekilde tekrar tekrar gelirse, model onu ezberleyebilir.
- Shuffle olmazsa **batch'ler dengesiz dağılabilir** (örneğin ilk 30 örnek sadece "dolu" olur).

🎁 2. Test Setinde `shuffle=False` Neden Önemli?

Test setinde genellikle:

- **Sonuçları analiz etmek** istersin (hangi görüntü doğru tahmin edilmiş, hangisi yanlış?)
- **Confusion matrix**, ROC curve, precision/recall gibi metrikleri hesaplarken, **gerçek etiketlerle tahminlerin sırası eşleşmeli.**

Eğer `shuffle=True` olursa:

- `model.evaluate()` yine doğru skor verir çünkü etiketler de karışır.
- **Ama `model.predict()` ile tahmin alırsan sıra karışır.**
- Ve bu durumda, tahminleri gerçek etiketlerle eşleştirmen çok zorlaşır.

Kısa Örnek:

```
python
Kodu kopyala
preds = model.predict(test_generator)
print(test_generator.filenames[:5])  # Eğer shuffle=False ise bu sırayla eşleşir.
```

Eğer `shuffle=True` olsaydı, bu eşleşme **tutarsız** olurdu → doğru analiz yapamazsın.

✅ Özet:

Set	shuffle	Neden
Train	True	Daha iyi genelleme, overfitting azalır
Val/Test	False	Tahminleri analiz etmek için sıra korunmalı

Not: TensorFlow'da doğrudan *Inception v1 (GoogLeNet)* yok. En yakın ve çok daha güçlü olanı: **InceptionV3.** Inception V3 yaklaşık 311 katman içerir.

Burada base modeli kendi modelimizin içinde kullandık ya, onun yerine base modeli sadece feature extractor olarak kullanabilir miydik? Eğer cevabın evetse o zaman da bu işleme transfer learning diyebilir miyiz? Bir de eğer base modeli feature extractor olarak kullanırsak sonrasında sınıflandırma aşamalarını tekrar bir network olarak oluşturmak yerine bir makine öğrenmesi modeli kullanabilir miydik?

✅ 1. Base modeli sadece feature extractor olarak kullanabilir miyiz?

Evet, kesinlikle kullanabilirsin.
Bu yöntemin adı genelde:

- **"Feature extraction with frozen base"** veya sadece

- **"Feature-based transfer learning"** olarak geçer.

Nasıl olur?

1. Pretrained modeli yüklersin (örneğin `InceptionV3`, `include_top=False`)
2. `model.predict()` ya da `model(x)` ile her görüntü için **özellik vektörleri (feature maps)** üretirsin.
3. Bu vektörleri kaydedersin.
4. Bu vektörleri bir klasik ML algoritmasıyla sınıflandırırsın (SVM, RandomForest, Logistic Regression vs.)

Bu yöntemde **base model sadece özellik çıkarır (forward pass)**. Eğitim sırasında değişmez.

✅ 2. Bu işleme yine transfer learning diyebilir miyiz?

Evet, %100 transfer learning kapsamına girer.

Transfer learning'in iki türü vardır:

Tür	***Açıklama***
Feature Extraction	Pretrained modelin çıktısını alır, üstüne yeni sınıflayıcı eklersin. Base model dondurulmuştur.
Fine-Tuning	Önceki yönteme ek olarak base modelin bazı katmanlarını da yeniden eğitirsin.

Sen sadece özellikleri alıp klasik bir sınıflayıcı (SVM vs.) kullanırsan bu da feature-based transfer learning'dir.

✅ 3. Son aşamada klasik makine öğrenmesi modeli kullanabilir miyiz?

Evet, bu oldukça yaygın bir uygulamadır.

Adımlar:

1. Görselleri `model.predict()` veya `model(x)` ile geçirip sabit boyutlu bir vektör elde et:

```
python
Kodu kopyala
feature_extractor = InceptionV3(weights='imagenet', include_top=False,
pooling='avg')
features = feature_extractor.predict(x_data)
```

2. Bu `features` array'ini `X`, etiketleri de `y` olarak al:

```
python
Kodu kopyala
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression()
clf.fit(X, y)
```

3. Tahmin yap:

```
python
Kodu kopyala
preds = clf.predict(X_test)
```

Avantajları:

- Çok daha hızlı eğitim (ML algoritmaları çok hızlı converge eder).
- Derin öğrenme bilmeyen takımlarla paylaşması kolaydır.
- Veri azsa daha kararlı sonuç verebilir.

Dezavantajı:

- Backpropagation yok → özellik çıkarıcı katmanlar optimize edilmez.
- End-to-end model olmaktan çıkar, parçalı sistem olur.

🔍 Örnek Uygulama Tipi

Yöntem	***Ne zaman kullanılır?***
Feature extraction + ML modeli	Veri azsa, hızlı sonuç isteniyorsa
Feature extraction + kendi dense head	Derin öğrenme altyapısı varsa, daha fazla veri varsa
Fine-tuning	Güçlü donanım + daha fazla veri + yüksek doğruluk hedefi varsa

Confusion Matrix (Karışıklık Matrisi), bir sınıflandırma modelinin **nerede doğru nerede yanlış tahmin yaptığını detaylı biçimde görmemizi sağlayan bir tablodur.**

🎯 Ne İşe Yarar?

- Sadece **accuracy** görmek yeterli değildir.
  Model 100 örneğin 95'ini doğru tahmin ettiyse bu güzel, **ama yanlış tahmin ettiği 5 taneyi nasıl dağıttı?**
- Confusion matrix sana **doğru ve yanlış sınıflandırmaları sınıf bazında gösterir.**

🔍 Yapı (Binary Classification için)

İki sınıf olduğunu varsayalım:

- 1 = **Araç (vehicle)**
- 0 = **Araç değil (non-vehicle)**

Confusion matrix şöyle görünür:

	Tahmin: Araç (1)	**Tahmin: Araç Değil (0)**
Gerçek: Araç (1)	TP (True Positive)	FN (False Negative)
Gerçek: Araç Değil (0)	FP (False Positive)	TN (True Negative)

□ Terimlerin Anlamı

- **TP (True Positive):** Gerçekten araç olanı doğru "araç" olarak tahmin etmiş.
- **TN (True Negative):** Gerçekten araç olmayanı doğru "araç değil" olarak tahmin etmiş.
- **FP (False Positive):** Araç olmayanı yanlışlıkla "araç" demiş.
- **FN (False Negative):** Gerçek araç olanı "araç değil" demiş (kaçırmış).

📈 Neden Önemli?

- **Precision**, **Recall**, **F1-Score** gibi metrikler bu tablo üzerinden hesaplanır.
- Modelin hangi sınıfta zayıf olduğunu bulmak için kullanılır.
- Örneğin:
    - Eğer çok fazla **FP** varsa: Model **araç olmayanlara fazla "araç" diyor.**
    - Eğer çok fazla **FN** varsa: Model **araçları kaçırıyor**, bu özellikle otonom sürüşte tehlikeli olur.

✅ Özet

Sağladığı Bilgi	**Neden Önemli?**
Hangi sınıfı ne kadar doğru bildi?	Dengesiz veri setlerinde accuracy yanıltıcı olabilir
Yanlışlar hangi sınıfta yoğunlaşıyor?	Modelin neyi öğrenemediğini gösterir
Hedeflenen sınıfı kaçırıyor mu?	Özellikle güvenlik kritik uygulamalarda önemli

🎯 Ne İşe Yarar Bu Metrikler?

Metrik	**Anlamı**
Precision	"Model araç dediğinde ne kadar isabetli?" → **Doğru araç tahminlerinin oranı**
Recall	"Gerçek araçların ne kadarını yakaladı?" → **Kaçırdığı araçları ölçer**
F1-Score	Precision ve recall'ün dengeli ortalaması

🔍 Metrik Açıklamaları

◆ Precision (Kesinlik)

“Model **araç dediğinde** ne kadar isabetli?”

$$Precision = TP / TP+FP$$

Yüksekse → Model fazla false positive yapmıyor.

◆ Recall (Duyarlılık)

“Model **gerçek araçları** ne kadar kaçırmadan buldu?”

Recall=TP / TP+FN

Yüksekse → Model araçları iyi yakalıyor, çok kaçırmıyor.

◆ **F1-Score**

Precision ve recall'ün dengeli ortalaması.
Özellikle sınıflar dengesizse (örneğin araçlar azınlıktaysa) F1-score çok anlamlıdır.

F1=2·(Precision·Recall / Precision+Recall)

`predict()` ve `evaluate()`, her ikisi de test verisi üzerinde modelin performansını ölçmeye yarar ama **amaçları ve çıktıları farklıdır.**

Aşağıda farklarını sade ve net şekilde açıklıyorum:

🔎 **`model.predict()`**

Ne yapar?

- Verilen veri üzerinde **tahmin** yapar.
- Çıktı: **Sınıf olasılıkları** veya **doğrudan tahmin değerleri** (örneğin sigmoid çıktısı: 0.87, 0.03…)

Ne için kullanılır?

- Tahminleri dışa aktarmak, görselleştirmek, confusion matrix hesaplamak, ROC eğrisi çizmek vs.
- Post-processing (örneğin `y_pred > 0.5` ile sınıf tahmini yapmak) için kullanılır.

Örnek:

```
python
KopyalaDüzenle
y_pred_probs = model.predict(test_generator)
y_pred = (y_pred_probs > 0.5).astype("int")
```

□ **`model.evaluate()`**

Ne yapar?

- Verilen veri üzerinde **kayıp fonksiyonu (loss)** ve **belirttiğin metrikleri** (accuracy, etc.) hesaplar.
- Çıktı: sayısal performans ölçütleri (örn. `[loss, accuracy]`)

Ne için kullanılır?

- Modelin genel başarımını hızlıca görmek için.
- Eğitim sonrası test değerlendirmesi yapmak için.

Örnek:

```
python
KopyalaDüzenle
loss, acc = model.evaluate(test_generator)
print(f"Test Accuracy: {acc:.4f}")
```

□ Kıyaslama Tablosu

Özellik	**predict()**	**evaluate()**
Ne yapar?	Sınıf tahmini üretir (olasılık/veri)	Kayıp ve metrik hesaplar
Ne döner?	y_pred_probs (örneğin: 0.88, 0.04...)	[loss, accuracy]
Kullanım amacı	Görselleştirme, analiz, ROC, confusion	Performans raporu, hızlı kıyas
Manuel iş gerekir mi?	Evet (y_pred > 0.5 gibi)	Hayır (otomatik hesaplar)

□ Ne zaman hangisini kullanmalıyım?

Amaç	**Hangisi?**
Test seti için accuracy görmek istiyorum	✅ evaluate()
Confusion matrix çizmek istiyorum	✅ predict()
ROC eğrisi çizmek istiyorum	✅ predict()
Sınıf olasılıklarını istiyorum	✅ predict()
Modeli hızlıca ölçmek istiyorum	✅ evaluate()

NOT -> Mevcut very setinde inceptionV3 modeliyle eğitim çok uzun sürüyor. Sadece 5 epoch'un tamamlanması bile yaklaşık 1 saat sürdü.

Fine-tuning işlemini yapmak için en doğru zaman:

🔹 **Ön eğitimi (sadece head eğitimi) başarıyla tamamladıktan hemen sonra**
🔹 Yani senin şu an bulunduğun nokta: **ideal zaman** ✅

🎯 Neden Şimdi?

- Modelin üst kısmı (Dense katmanlar) zaten eğitildi.
- Base model (InceptionV3) şu ana kadar **dondurulmuştu**, yani sadece genel özellikler kullanıldı.
- Şimdi, base modelin son birkaç katmanını açarak, bu ağı **senin veri setine özel olarak biraz daha hassaslaştırabiliriz.**
- Bu, genellikle doğrulukta artış sağlar (özellikle validation/test skorları yakınsa).

⚠️ Not:

- **Eğitim biraz daha yavaş sürebilir**, çünkü artık daha fazla parametre güncellenecek.
- `EarlyStopping` veya `ModelCheckpoint` gibi callback'ler de istersen ekleyebiliriz.

Sadece son katmanları eğitime açmak **bilinçli ve mantıklı bir tercihtir** — sebebi hem teknik hem de pratik gerekçelere dayanır.

Aşağıda adım adım açıklıyorum:

✅ 1. Erken Katmanlar Genel Özellikler Öğrenir

- Convolutional katmanlar derinleştikçe öğrenilen özellikler daha soyut hale gelir:
  - **İlk katmanlar**: kenar, renk, çizgi, doku
  - **Orta katmanlar**: parça-parça yapılar (tekerlek, cam gibi)
  - **Son katmanlar**: sınıf spesifik yapılar (araba vs değil ayırımı)

Dolayısıyla, ilk katmanları eğitmeye gerek yok çünkü bu özellikler **tüm görsel görevlerde ortaktır** ve zaten iyi öğrenilmiştir.

✅ 2. Daha Az Hesaplama – Daha Az Overfitting Riski

- Tüm modeli açarsan:
    - Eğitim çok yavaşlar,
    - Parametre sayısı fırlar,
    - Küçük veri setlerinde **overfitting riski ciddi artar.**

Sadece son birkaç katmanı açarak:

- Hesaplama yükünü sınırlarsın
- Gereksiz ağırlık değişimini önlersin
- İnce ayar (fine-tuning) yapmış olursun

✅ 3. Learning Rate ile Uyumlu Ayarlama

- Fine-tuning aşamasında kullanılan **küçük learning rate**, sadece birkaç katmanda işe yarar.
- Tüm modeli açarsan, bu küçük oran tüm ağı optimize edemez → sonuç kötüleşebilir.

🔍 Ne Zaman Tüm Katmanlar Açılır?

- Büyük veri setin varsa
- Modelin baştan sona yeni göreve uyum sağlaması gerekiyorsa
- Hesaplama gücün yeterliyse
  → Tüm katmanları açıp küçük learning rate ile yeniden eğitmek mantıklıdır.

Ama:

Senin gibi **transfer learning** yapan birinde en iyi yöntem:
✅ "Önce sadece head eğit, sonra son birkaç katmanı aç ve fine-tune et."

Özet:

Katman Türü	Eğitilmeli mi?	Neden?
İlk (erken) Conv katmanları	❌	Genel özellikler öğrenir, zaten iyi durumda
Son birkaç Conv bloğu	✅	Göreve özel ayarlama için
Dense (eklediğimiz head)	✅	Yeni görev için öğrenilmeli