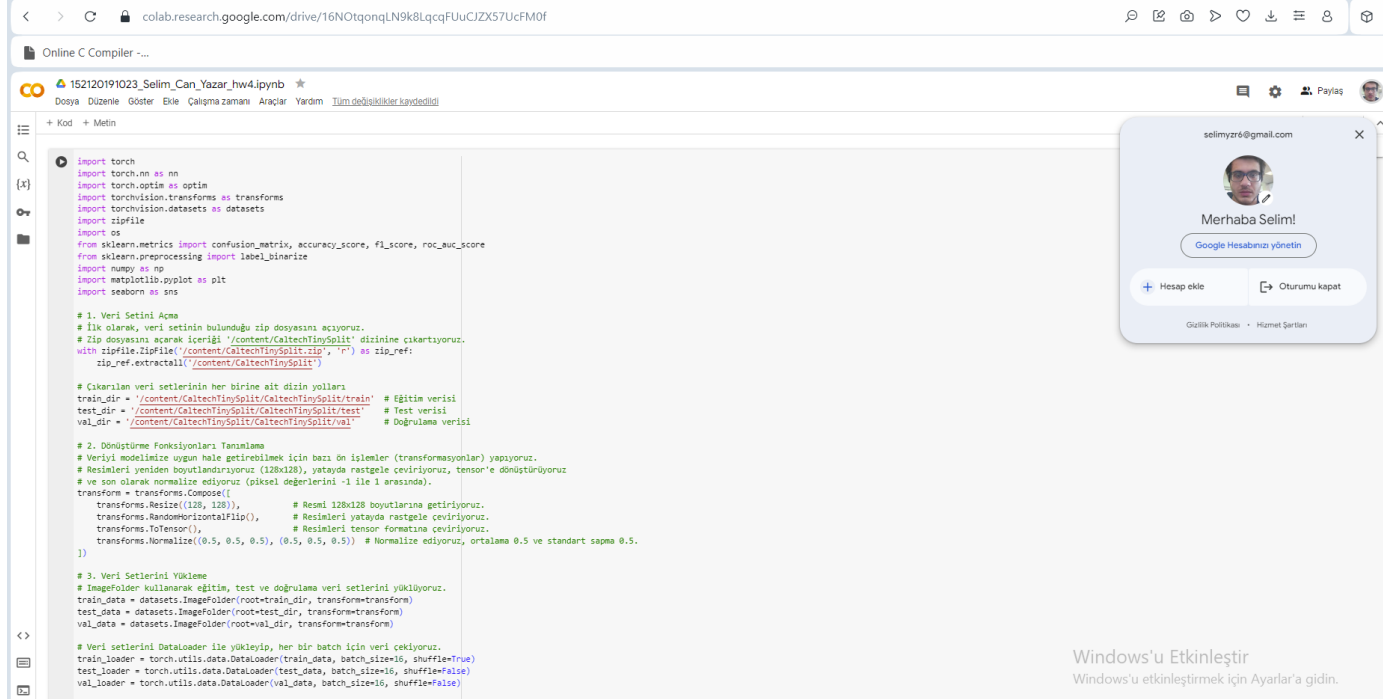


Deep Learning Hw4 Raporu

1. Gerekli Kütüphanelerin ve CaltechTinySplit Veri Setinin Yüklenmesi



```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
import torchvision.datasets as datasets
import zipfile
import os
from sklearn.metrics import confusion_matrix, accuracy_score, f1_score, roc_auc_score
from sklearn.preprocessing import label_binarize
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# 1. Veri Setini Açma
# İlk olarak, veri setinin bulunduğu zip dosyasına erişiyoruz.
# Zip dosyasını açarak içeriği '/content/CaltechTinySplit' dizinine çıkartıyoruz.
with zipfile.ZipFile('/content/CaltechTinySplit.zip', 'r') as zip_ref:
    zip_ref.extractall('/content/CaltechTinySplit')

# Çıkarılan veri setlerinin her birine ait dizin yolları
train_dir = '/content/CaltechTinySplit/CaltechTinySplit/train' # Eğitim verisi
test_dir = '/content/CaltechTinySplit/CaltechTinySplit/test' # Test verisi
val_dir = '/content/CaltechTinySplit/CaltechTinySplit/val' # Doğrulama verisi

# 2. Dönüştürme Fonksiyonları Tanımlama
# Veriyi modelimize uygun hale getirebilmek için bazı ön işlemler (transformasyonlar) yapıyoruz.
# Resimleri yeniden boyutlandırıyoruz (128x128), yataıda rastgele çeviriyoruz, tensor'e dönüştürüyoruz
# ve son olarak normalize ediyoruz (piksel değerlerini -1 ile 1 arasında).
transform = transforms.Compose([
    transforms.Resize((128, 128)), # Resmi 128x128 boyutlarına getiriyoruz.
    transforms.RandomHorizontalFlip(), # Resimleri yataıda rastgele çeviriyoruz.
    transforms.ToTensor(), # Resimleri tensor formatına çeviriyoruz.
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) # Normalize ediyoruz, ortalama 0.5 ve standart sapma 0.5.
])

# 3. Veri Setlerini Yükleme
# ImageFolder kullanarak eğitim, test ve doğrulama veri setlerini yükliyoruz.
train_data = datasets.ImageFolder(root=train_dir, transform=transform)
test_data = datasets.ImageFolder(root=test_dir, transform=transform)
val_data = datasets.ImageFolder(root=val_dir, transform=transform)

# Veri setlerini DataLoader ile yükleyip, her bir batch için veri çekiyoruz.
train_loader = torch.utils.data.DataLoader(train_data, batch_size=16, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=16, shuffle=False)
val_loader = torch.utils.data.DataLoader(val_data, batch_size=16, shuffle=False)
```

İlk olarak gerekli kütüphaneler import edilerek koda başlandı. Pytorch'un temel modüllerini torch ve torch.nn kütüphaneleri sağladı. Torch.optim kütüphanesi ise temel optimizasyon fonksiyonlarını içeriyor. Torchvision.transform ve torchvision.dataset kütüphaneleri de veriyi önyükleme ve yükleme görevlerini yerine getirdi. Modelin performansını öğrenmek için auc skoru, f1 skoru ,accuracy ve confusion matrisi de sklearn.metrics kütüphanesi ile elde ettik. Python'da klasik olarak matematiksel işlemleri yapmak için numpy , grafikleri elde edip daha iyi bir görsellik elde etmek içinde matplotlib.pyplot ve seaborn kütüphanelerini kullandık.

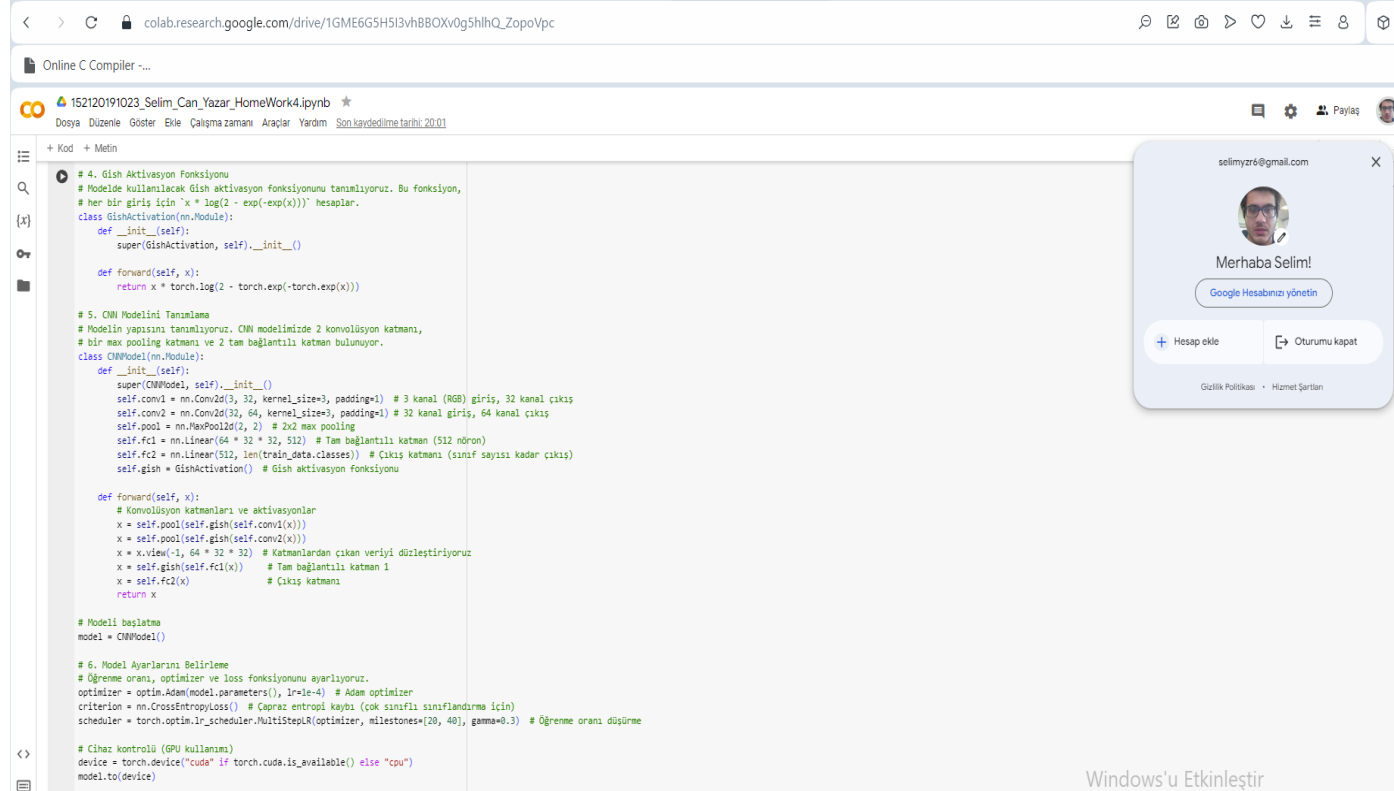
Caltech veri setimizi Google Colaba yükleyebilmek için önce .zip dosyası haline getirdik daha sonra manuel olarak arayüzden yükledik. Google Colab da zipli dosyalar okunamadığından Zipfile.ZipFile zipfile kütüphanesindeki ilgili fonksiyonun yardımıyla veri setimizi .zip ten /content/CaltechTinySplit veri yoluna çıkarmayı başardık. İlgili veri yoluna dosyamızı çıkardıktan sonra içerisindeki train, test ve val klasörlerini ilgili veri yollarında sınıflandırır.

CNN modelimizi eğitmeye başlamadan verileri modelle uyumlu hale getirmeliyiz. Transform.Compose transform kütüphanesinin Compose fonksiyonuyla veri seti modele uygun hale getirildi. Transform kütüphanesinin Resize fonksiyonuyla veriler

128*128 piksel olacak şekilde boyutlandırıldı. Modelimizin verilerin yatay simetrisini öğrenmesi için transform kütüphanesinin RandomHorizontalFlip fonksiyonu ile veriler random bir şekilde yatay hale getirildi. ToTensor fonksiyonuyla ise Pytorch kütüphanesinin tensor formatına uygun hale getirilerek CNN modeline uygun hale getirdi. Son adımda ise verilerin piksel değerleri -1 ile 1 aralığına çekilerek modelin daha stabil bir şekilde öğrenmesi amaçlandı.

Veri setimizin klasör yapısındaki verileri düzenleyip etiketlendirmek için torchvision.datasets modülünde bulunan ImageFolder sınıfını kullanarak veri setimizin train , test ve val kısımlarını yükledik. DataLoader ile yüklenen veri seti modelin her eğitim adımında kullanılmaya müsait bir hale getirildi. `batch_size=16` parametresi olarak ayarlanarak, veriler her adımda 16'şar parçalık gruplar halinde modele verildi. Eğitim verileri rastgele sıralanırken shuffle parametresi (shuffle=True) olarak , test ve doğrulama verileri ise (shuffle=False) parametresi ile sıralı olarak modele verilir. Bu yöntemle verilerin modele belirli bir düzende ve doğru boyutlarda aktarılması sağlandı ve modelin performans değerlendirmesini mümkün hale getirildi.

2. Gish Aktivasyon Fonksiyonu ve CNN Modeli



```
# 4. Gish Aktivasyon Fonksiyonu
# Modelde kullanılacak Gish aktivasyon fonksiyonunu tanımlıyoruz. Bu fonksiyon,
# her bir giriş için 'x * log(2 - exp(-exp(x)))' hesaplar.
class GishActivation(nn.Module):
    def __init__(self):
        super(GishActivation, self).__init__()

    def forward(self, x):
        return x * torch.log(2 - torch.exp(-torch.exp(x)))

# 5. CNN Modelini Tanımlama
# Modelin yapısını tanımlıyoruz. CNN modelimizde 2 konvolüsyon katmanı,
# bir max pooling katmanı ve 2 tam bağlantılı katman bulunuyor.
class CNNModel(nn.Module):
    def __init__(self):
        super(CNNModel, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1) # 3 kanal (RGB) giriş, 32 kanal çıkış
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1) # 32 kanal giriş, 64 kanal çıkış
        self.pool = nn.MaxPool2d(2, 2) # 2x2 max pooling
        self.fc1 = nn.Linear(64 * 32 * 32, 512) # Tam bağlantılı katman (512 nöron)
        self.fc2 = nn.Linear(512, len(train_data.classes)) # Çıkış katmanı (sınıf sayısı kadar çıkış)
        self.gish = GishActivation() # Gish aktivasyon fonksiyonu

    def forward(self, x):
        # Konvolüsyon katmanları ve aktivasyonlar
        x = self.pool(self.gish(self.conv1(x)))
        x = self.pool(self.gish(self.conv2(x)))
        x = x.view(-1, 64 * 32 * 32) # Katmanlardan çıkan veriyi düzleştiriyoruz
        x = self.gish(self.fc1(x)) # Tam bağlantılı katman 1
        x = self.fc2(x) # Çıkış katmanı
        return x

# Modeli başlatma
model = CNNModel()

# 6. Model Ayarlarını Belirleme
# Öğrenme oranı, optimizör ve loss fonksiyonunu ayarlıyoruz.
optimizer = optim.Adam(model.parameters(), lr=1e-4) # Adam optimizör
criterion = nn.CrossEntropyLoss() # Çapraz entropi kaybı (çok sınıflı sınıflandırma için)
scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, milestones=[20, 40], gamma=0.3) # Öğrenme oranı düşürme

# Cihaz kontrolü (GPU kullanımı)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

1. Gish Aktivasyon Fonksiyonu

Gish fonksiyonu bu modelin özel bir aktivasyon fonksiyonu olarak kullanılır. Bu fonksiyon her seferinde $(x) \ln(2 - \exp(-\exp(x)))$ dönüşümünü gerçekleştirir. Gish aktivasyonu, normal aktivasyon fonksiyonlarıyla karşılaştırıldığında, modelin daha doğrusal olmayan korelasyonları yakalamasını etkili bir şekilde sağlar. Ancak, bu aktivasyonun modele dahil edilmesi, karmaşık ilişkilerin daha iyi temsil edilmesiyle sonuçlanır.

2. CNN Model Yapısı

Bu modelde, görüntü verilerini işlemek için Convolutional Sinir Ağı (CNN) kullanılır. CNN modeli, genellikle yararlı görüntü özelliklerini çıkarma amacıyla konvolüsyon (conv) katmanlarından oluşur. Modelde aşağıdaki yapılar kullanılır:

2.1. Konvolüsyon Katmanları: İki konvolüsyon katmanı kullanılır. Giriş RGB'sine (3 kanal) bir tanesi ulaşır ve ardından çıkış kanallarına 32 kanal ulaşır. İkinci katman, maksimum 64 çıkış kanalına genişletilmiş 32 giriş kanalından oluşur. Bu katmanlar, bir görüntünün belirli yönlerini tanımlayan özellik haritaları üretir.

2.2. Maksimum Havuzlama Katmanı: Her konvolüsyon katmanından sonra bir maksimum havuzlama gerçekleştirilir. Max pooling daha sonra özellik haritasındaki maksimum değeri alır, aşağı örnekleme yapar ve bilgi kaybını azaltır. Sonuç olarak, model yalnızca bilgisayar kaynaklarından tasarruf etmekle kalmaz, aynı zamanda ayırt edici özelliklere de odaklanır.

2.3. Tam Bağlantılı Katmanlar: Aşağıdaki konvolüsyon ve havuzlama katmanlarının çıktıları daha sonra düzleştirilir ve ağın tam bağlantılı katmanlarının girişi olarak kullanılır. İlk tam bağlantılı katman, 512 nöron ve ardından bir aktivasyon fonksiyonuna sahip özelliklerden oluşan bu düzleştirilmiş girişi giriş olarak alır. Çıkış katmanında, çıkış sayısı sınıf sayısına göre normalleştirilir, ardından sınıflandırma yapılır.

2.4. Gish Aktivasyon Fonksiyonu: Gish aktivasyonu her konvolüsyon ve her tam bağlantılı katmandan sonra kullanılır ve doğrusal olmayan özelliklerin öğrenilmesini geliştirir.

3. Model Ayarları

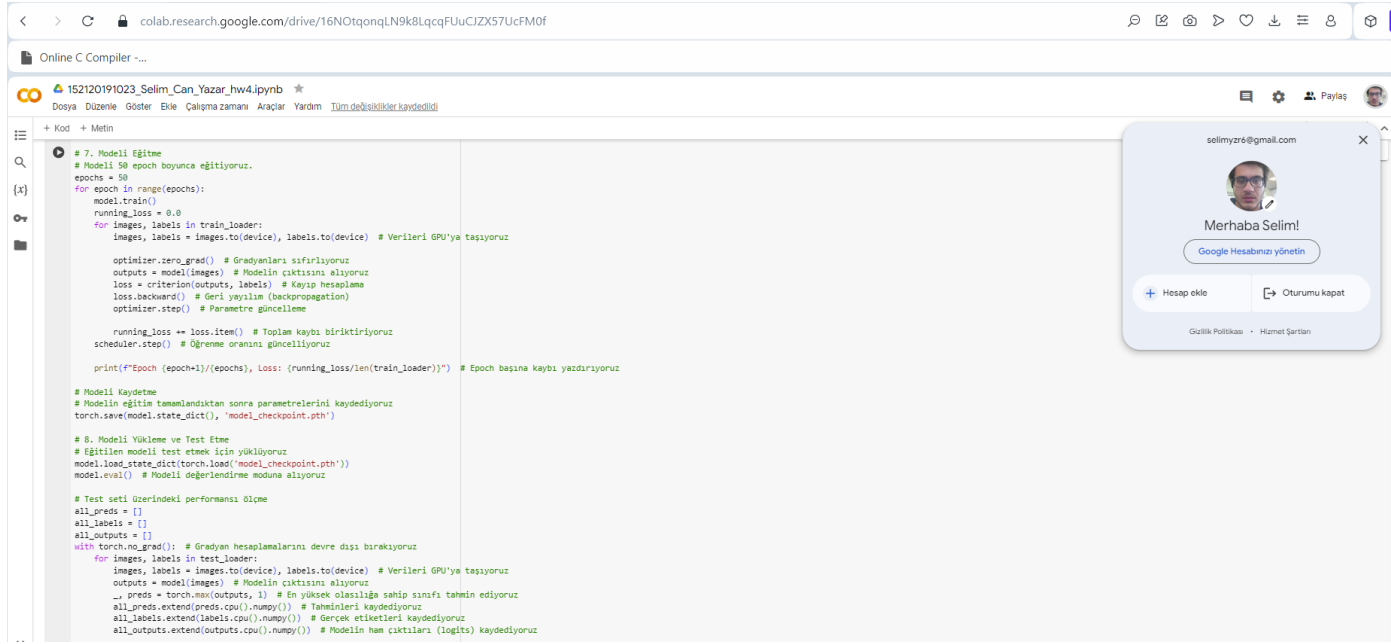
Öğrenme oranı, optimizasyon prosedürü ve kayıp fonksiyonu gibi kritik model parametreleri model spesifikasyonunda seçilir.

- **Optimizer (Adam):** Adam optimizasyon algoritması, modelin parametrelerini yinelemeli olarak güncellemek için kullanılır. Adam, özellikle büyük veri kümelerinde yüksek hız/verimlilik optimizasyonudur.

- **Loss Function (Çapraz Entropi Kaybı):** Çapraz entropi kaybı, model sınıflandırma performans ölçütü olarak kullanılır. Çok sınıflı sınıflandırmada en popüler tekniklerden biri olarak, bu, modelin doğru sınıfa olan mesafesini etkili bir şekilde ölçebilen bir yöntemdir.

- **Learning Rate Scheduler:** Öğrenme oranı, eğitim sırasında bir zamanlama aracılığıyla izlenir ve uyarlanabilir şekilde güncellenir. Model eğitimi, öğrenme oranını belirli epoch sayılarında (Örnekte 50) (örneğin, 20 ve 40) düşürerek sabitlenir. Modelin öğrenme oranının yüksek olduğu başlangıçta çok hızlı öğrenmesini ve ardından daha yavaş bir hızda öğrenmesini sağlar.

3. Model Eğitimi ve Modeli Kaydetme



```
# 7. Modeli Eğitme
# Model 50 epoch boyunca eğitiyoruz.
epochs = 50
for epoch in range(epochs):
    model.train()
    running_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device) # Verileri GPU'ya taşıyoruz

        optimizer.zero_grad() # Gradyanları sıfırlıyoruz
        outputs = model(images) # Modelin çıktısını alıyoruz
        loss = criterion(outputs, labels) # Kayıp hesaplama
        loss.backward() # Geri yayılım (backpropagation)
        optimizer.step() # Parametre güncelleme

    running_loss += loss.item() # Toplam kaybı biriktiriyoruz
    scheduler.step() # Öğrenme oranını güncelliyoruz

    print(f"Epoch {epoch+1}/{epochs}, Loss: {running_loss/len(train_loader)}") # Epoch başına kaybı yazdırıyoruz

# Modeli Kaydetme
# Modelin eğitim tamamlandıktan sonra parametrelerini kaydediyoruz
torch.save(model.state_dict(), 'model_checkpoint.pth')

# 8. Modeli Yükleme ve Test Etme
# Eğitilen model test etmek için yükliyoruz
model.load_state_dict(torch.load('model_checkpoint.pth'))
model.eval() # Modeli değerlendirme moduna alıyoruz

# Test seti üzerindeki performansı ölçme
all_preds = []
all_labels = []
all_outputs = []
with torch.no_grad(): # Gradyan hesaplamalarını devre dışı bırakıyoruz
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device) # Verileri GPU'ya taşıyoruz
        outputs = model(images) # Modelin çıktısını alıyoruz
        _, preds = torch.max(outputs, 1) # En yüksek olasılığa sahip sınıfı tahmin ediyoruz
        all_preds.extend(preds.cpu().numpy()) # Tahminleri kaydediyoruz
        all_labels.extend(labels.cpu().numpy()) # Gerçek etiketleri kaydediyoruz
        all_outputs.extend(outputs.cpu().numpy()) # Modelin ham çıktıları (logits) kaydediyoruz
```

1. Modeli Eğitme

Modelin eğitim sürecinde veri setinin train bölümü modelden 50 kez geçirilerek (50 epoch boyunca) modelin her seferinde biraz daha öğrenmesi sağlanır. Train aşamasında ilk olarak model `train()` fonksiyonu ile train moduna alınır. Train modunda model, dropout ve batch normalization gibi işlemleri etkinleştirir. Bu işlemler sayesinde modelin öğrenme kapasitesini artırmak için kullanırız. Her epoch başında o epoch'un toplam kayıp miktarını tutacak olan `running_loss` değişkeni sıfırlanır. Eğitim veri kümesindeki her örneği işlerken, modelin parametrelerinin güncellenmesini sağlayacak gradyanların birikmemesi için her iterasyonda `optimizer.zero_grad()` komutu ile gradyanlar sıfırlanır.

Modelin ileri yönde geçişi (`forward pass`), yani tahmin yapma süreci `outputs = model(images)` komutu ile başlar. Bu aşamada model, giriş görüntüleri üzerinde tahmin yapar ve `criterion(outputs, labels)` komutu ile tahminlerin gerçek etiketlere göre ne kadar hata içerdiği, yani kaybı hesaplanır. Ardından, geri yayılım (`backward pass`) aşamasında `loss.backward()` fonksiyonu ile modelin ağırlıkları üzerindeki gradyanlar hesaplanır; bu gradyanlar, modelin öğrenme sürecinin temelini oluşturur. `optimizer.step()` komutu ise bu gradyanlar doğrultusunda modelin ağırlıklarını güncelleyerek hataları minimize etmeye çalışır. Epoch sonuna kadar her iterasyondaki kayıp `running_loss` ile biriktirilir ve epoch sonunda ortalama kayıp değerini konsola yazdırarak modelin performansını buradan takip etmemizi sağlar. `scheduler.step()` fonksiyonu ile belirlenen epoch'larda öğrenme oranı azaltılır ve böylece eğitim süreci ilerledikçe modelin daha küçük adımlarla güncellenmesi sağlanır.

Modeli Kaydetme

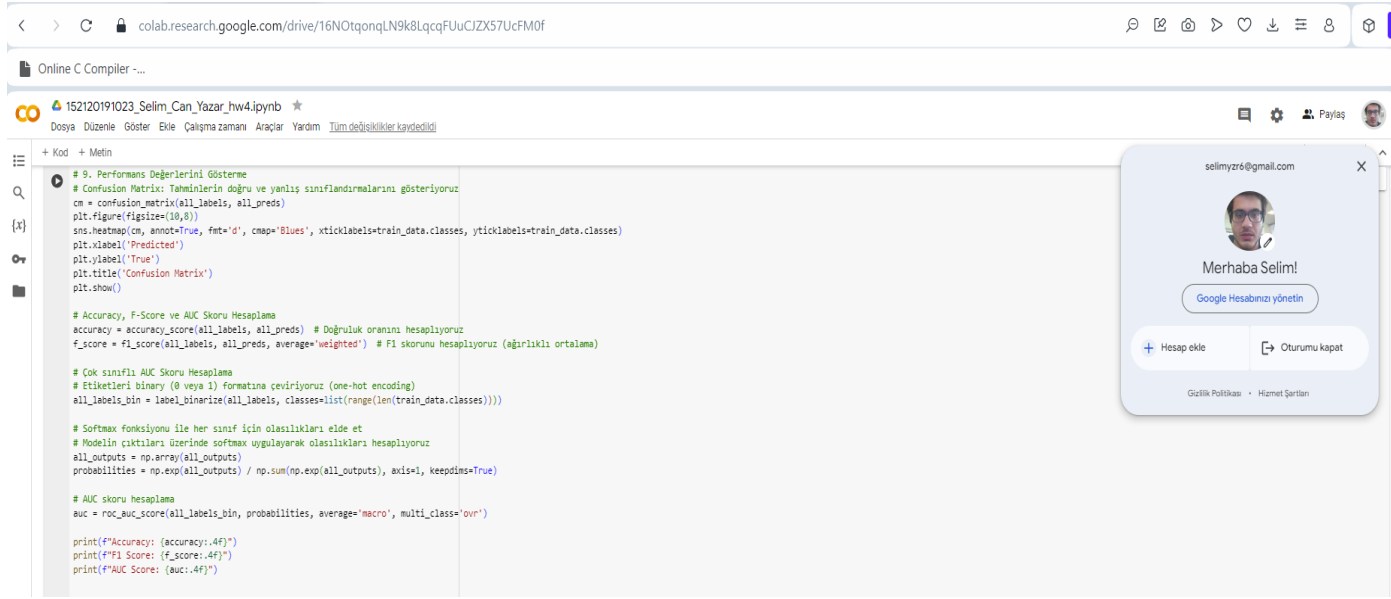
Eğitim süreci tamamlandıktan sonra, modelin öğrendiği parametreler ``torch.save(model.state_dict(), 'model_checkpoint.pth')`` komutu ile kaydedilir. Bu dosya, modelin eğitilmiş ağırlıklarını içerir ve bu sayede modelin daha sonraki süreçlerde yeniden eğitilmesine veya test edilmesine olanak sağlar. Bu yöntemle eğitilmiş modelin parametreleri yedeklenmiş olur ve gerektiğinde tekrar kullanılabilir.

2. Modeli Yükleme ve Test Etme

Eğitim tamamlandıktan sonra modelin test verisindeki performansını ölçmek için ``model.load_state_dict(torch.load('model_checkpoint.pth'))`` komutu ile kaydedilen model parametreleri yüklenir. ``model.eval()`` komutu ise modelin değerlendirme moduna geçirilmesini sağlar; böylece eğitimde kullanılan dropout gibi işlemler devre dışı bırakılarak daha istikrarlı bir performans elde edilir. Test sürecinde gradyan hesaplamaları (``torch.no_grad()``) fonksiyonu ile kapatılır çünkü model yalnızca tahmin yapmak için kullanılacak ve bu sayede bellek kullanımı azalacak.

Veri setinin test kısmında modelin performansı ölçülürken her test örneği modelden tek tek geçirilir ve tahmin edilen sınıf ``torch.max(outputs, 1)`` fonksiyonu ile belirlenir. Modelin tahminleri ``all_preds`` listesine kaydedilirken gerçek etiketler ``all_labels`` listesinde tutulur. Buna ek olarak modelin her örnek için ürettiği ham tahmin değerleri de ``all_outputs`` listesine kaydedilir. Bu veriler aracılığıyla modelin doğruluk gibi performans metriklerinin hesaplanması sağlanır. Test süreci, modelin eğitimde öğrendiği bilginin yeni veriler üzerindeki başarısını değerlendirmek için en önemli aşamadır.

4. Confusion matrix , AUC skoru , F1 skoru, Accuracy



```
# 9. Performans Değerlerini Gösterme
# Confusion Matrix: Tahminlerin doğru ve yanlış sınıflandırılmalarını gösteriyoruz
cm = confusion_matrix(all_labels, all_preds)
plt.figure(figsize=(10,8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=train_data.classes, yticklabels=train_data.classes)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()

# Accuracy, F-Score ve AUC Skoru Hesaplama
accuracy = accuracy_score(all_labels, all_preds) # Doğruluk oranını hesaplıyoruz
f_score = f1_score(all_labels, all_preds, average='weighted') # F1 skorunu hesaplıyoruz (ağırlıklı ortalama)

# Çok sınıflı AUC Skoru Hesaplama
# Etiketleri binary (0 veya 1) formatına çeviriyoruz (one-hot encoding)
all_labels_bin = label_binarize(all_labels, classes=list(range(len(train_data.classes))))

# Softmax fonksiyonu ile her sınıf için olasılıkları elde et
# Modelin çıktıları üzerinde softmax uygulayarak olasılıkları hesaplıyoruz
all_outputs = np.array(all_outputs)
probabilities = np.exp(all_outputs) / np.sum(np.exp(all_outputs), axis=1, keepdims=True)

# AUC skoru hesaplama
auc = roc_auc_score(all_labels_bin, probabilities, average='macro', multi_class='ovr')

print(f"Accuracy: {accuracy:.4f}")
print(f"F1 Score: {f_score:.4f}")
print(f"AUC Score: {auc:.4f}")
```

1. Confusion Matrix ile Performans Gösterimi

Modelin tutarlılığını kullanıcıya anlaşılır bir şekilde göstermek için öncelikle `confusion_matrix` fonksiyonu kullanılarak modelin tahminlerinin (prediction) etiketleri ile gerçek verilerin etiketlerin karşılaştırıldığı bir "confusion matrix" (karmaşıklık matrisi) oluştururuz. Bu matris sayesinde modelin her sınıf için doğru ve yanlış sayısını tablo şeklinde kullanıcının önüne koymuş olduk. Örnek vermek gerekirse her bir satırda ilgili sınıfın gerçek etiketleri (Örneğin camera ,cannon ,flamingo,pizza, cellphone vb.), sütunda ise modelin sınıflar için tahmin ettiği etiketler gösterilir. Bu sayede, modelin hangi sınıflarda başarılı ya da başarısız olduğunu gözlemlemek kolaylaşır. Confusion Matris Seaborn kütüphanesinin `heatmap` fonksiyonu ile görselleştirilir. Elde edilen tablo sayesinde kullanıcıya her bir sınıf için modelin ne kadar iyi tahmin yaptığını renk yoğunluğuna göre kolayca anlamasını sağlar. Tabloda ilgili hücredeki renk yoğunluğu arttıkça tutarlı tahminlerin sayısındaki artış anlatılmış olur. Grafiğin eksenlerinde, `train_data.classes` ile sınıf isimleri eklenerek hangi sınıflara karşılık geldiği belirtilir.

2. Doğruluk (Accuracy) ve F1 Skorunu Hesaplama

Ardından, modelin genel başarısını gösteren `accuracy` (doğruluk) ve `f_score` (F1 skoru) metrikleri hesaplanır. `accuracy_score` fonksiyonu, modelin tüm test örneklerindeki doğru sınıflandırma oranını verir ve bu değer, modelin ne kadar doğru tahmin yaptığını tek bir sayı ile ifade eder. F1 skoru ise doğruluk ve duyarlılığı birleştirerek modelin dengesini ölçer. Bu şekilde `average='weighted'` parametresi ile

sınıflar arasındaki dengesizlik göz önünde bulundurularak her sınıfın önem derecesi ağırlıklı olarak hesaba katılır. F1 skoru, özellikle veri setinde sınıfların verilerinde dengesizlik olduğu durumlarda modelin her bir sınıf için ne kadar başarılı olduğunu gösteren daha kapsayıcı bir değer üretir.

3.Çok Sınıflı AUC Skorunu Hesaplama

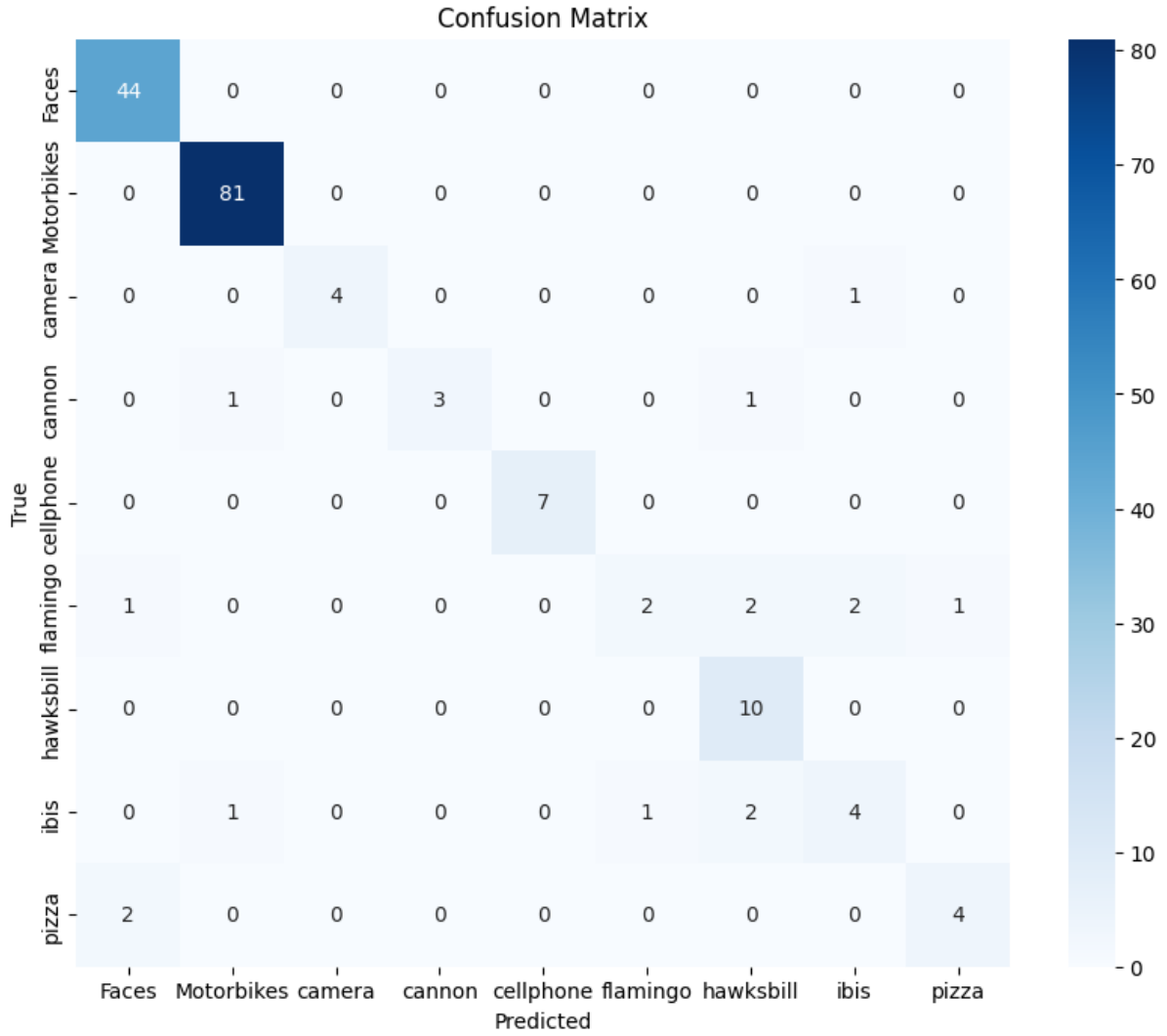
AUC (Area Under Curve) skoru sayesinde modelin her sınıf için olasılıkları ne kadar doğru tahmin ettiğini gösterir. AUC hesaplaması yapılırken, çok sınıflı bir yapı olduğundan önce `label_binarize` fonksiyonu kullanılarak sınıflar "one-hot" biçimine dönüştürülür. Bu işlem her sınıf için ayrı bir ikili vektör oluşturarak çok sınıflı veriyi AUC hesaplamasına uygun hale getirir.

Son olarak, modelin ham çıktılarına softmax fonksiyonu uygulanarak her sınıf için ayrı tahmin olasılıkları elde edilir. `np.exp(all_outputs) / np.sum(np.exp(all_outputs), axis=1, keepdims=True)` ifadesi sayesinde softmax fonksiyonu ile her sınıfın çıkış skorlarını olasılıklara dönüştürürüz. Bu olasılıklar aracılığıyla sınıflar arasındaki kesinlik derecesini değerlendirirken kullanırız. `roc_auc_score` fonksiyonu ile bu olasılıklar kullanılarak çok sınıflı "one-vs-rest" yöntemi ile AUC skoru hesaplanır. Sonuçta, `average='macro'` parametresiyle tüm sınıfların ortalama AUC skoru alınır.

4.Performans Sonuçlarını Görüntüleme İşlemleri

Hesaplanan doğruluk, F1 skoru ve AUC skoru print fonksiyonları ile output olarak yazdırılır, böylece modelin genel başarımı daha anlaşılır bir biçimde ifade edilmiş olur. Bu sonuçlar modelin hem sınıflar arasındaki başarısını hem de genel doğruluğunu değerlendirmek için kullanıcıya yararlı bilgiler sağlar.


```
Epoch 1/50, Loss: 0.7906811789554709
Epoch 2/50, Loss: 0.44861209914362166
Epoch 3/50, Loss: 0.3444030827130465
Epoch 4/50, Loss: 0.27451662072254457
Epoch 5/50, Loss: 0.19827147874542894
Epoch 6/50, Loss: 0.15694567826268374
Epoch 7/50, Loss: 0.1419673021444503
Epoch 8/50, Loss: 0.12688946027500445
Epoch 9/50, Loss: 0.09382564337814556
Epoch 10/50, Loss: 0.06769000623676488
Epoch 11/50, Loss: 0.04180325073128402
Epoch 12/50, Loss: 0.0328242709122338
Epoch 13/50, Loss: 0.023042264173700994
Epoch 14/50, Loss: 0.014954571659087807
Epoch 15/50, Loss: 0.013586731461862869
Epoch 16/50, Loss: 0.008202354594729566
Epoch 17/50, Loss: 0.005547238781411365
Epoch 18/50, Loss: 0.004864445197745227
Epoch 19/50, Loss: 0.0039166919163750105
Epoch 20/50, Loss: 0.0034111032935256577
Epoch 21/50, Loss: 0.0024664832705799627
Epoch 22/50, Loss: 0.0023267901647533802
Epoch 23/50, Loss: 0.0021782871693550774
Epoch 24/50, Loss: 0.002009876618296189
Epoch 25/50, Loss: 0.001889707705905654
...
Epoch 47/50, Loss: 0.0007909455694008262
Epoch 48/50, Loss: 0.0007420245395812398
Epoch 49/50, Loss: 0.000711382304528824
Epoch 50/50, Loss: 0.0006953274549339446
```



Accuracy: 0.9138
F1 Score: 0.9040
AUC Score: 0.9895

CNN Modelinin Gish aktivasyon fonksiyonu ve belirtilen parametreler içindeki performansı %91,38 ile çok yüksekti, bu da modelin test verilerinin çoğunu doğru bir şekilde tahmin ettiğini gösteriyor. Modelin %90,40'lık bir F1 puanı ile gerçek pozitif tahminler ve sınıflar arasında iyi bir denge sağladığını gördük, model ise %98,95'lik bir AUC puanı ile sınıflar arasında mükemmel bir şekilde ayrım yapabildi. Bu sonuçlar arasında, modelin genel olarak gerçekten iyi performans gösterdiğini ve mümkün olduğunca çok sayıda doğru sınıflandırma yaptığını gösterdik.