KARADENİZ TEKNİK ÜNİVERSİTESİ
MÜHENDİSLİK FAKÜLTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ

Derin Kapsül Ağları ile Yüz Tanıma Çalışması

BİTİRME PROJESİ

Adı SOYADI
EKREM AĞCA
FEYZANUR MEMİŞ
KADER NUR TEKİN
MERVE AYDIN
DANIŞMAN: DR. ÖĞR.ÜYESİ MURAT AYKUT

2021-2022 BAHAR DÖNEMİ

KARADENİZ TEKNİK ÜNİVERSİTESİ
MÜHENDİSLİK FAKÜLTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ

Derin Kapsül Ağları ile Yüz Tanıma Çalışması

BİTİRME PROJESİ

Adı SOYADI
EKREM AĞCA
FEYZANUR MEMİŞ
KADER NUR TEKİN
MERVE AYDIN
DANIŞMAN: DR. ÖĞR.ÜYESİ MURAT AYKUT

2021-2022 BAHAR DÖNEMİ

IEEE Etik Kuralları
IEEE Code of Ethics

Mesleğime karşı şahsi sorumluluğumu kabul ederek, hizmet ettiğim toplumlara ve üyelerine en yüksek etik ve mesleki davranışta bulunmaya söz verdiğimi ve aşağıdaki etik kurallarını kabul ettiğimi ifade ederim:

1. Kamu güvenliği, sağlığı ve refahı ile uyumlu kararlar vermenin sorumluluğunu kabul etmek ve kamu veya çevreyi tehdit edebilecek faktörleri derhal açıklamak;
2. Mümkün olabilecek çıkar çatışması, ister gerçekten var olması isterse sadece algı olması, durumlarından kaçınmak. Çıkar çatışması olması durumunda, etkilenen taraflara durumu bildirmek;
3. Mevcut verilere dayalı tahminlerde ve fikir beyan etmelerde gerçekçi ve dürüst olmak;
4. Her türlü rüşveti reddetmek;
5. Mütenasip uygulamalarını ve muhtemel sonuçlarını gözeterek teknoloji anlayışını geliştirmek;
6. Teknik yeterliliklerimizi sürdürmek ve geliştirmek, yeterli eğitim veya tecrübe olması veya işin zorluk sınırları ifade edilmesi durumunda ancak başkaları için teknolojik sorumlulukları üstlenmek;
7. Teknik bir çalışma hakkında yansız bir eleştiri için uğraşmak, eleştiriyi kabul etmek ve eleştiriyi yapmak; hatları kabul etmek ve düzeltmek, diğer katkı sunanların emeklerini ifade etmek;
8. Bütün kişilere adilane davranmak; ırk, din, cinsiyet, yaş, milliyet, cinsi tercih, cinsiyet kimliği veya cinsiyet ifadesi üzerinden ayırımcılık yapma durumuna girişmemek;
9. Yanlış veya kötü amaçlı eylemler sonucu kimsenin yaralanması, mülklerinin zarar görmesi, itibarlarının veya istihdamlarının zedelenmesi durumlarının oluşmasından kaçınmak;
10. Meslektaşlara ve yardımcı personele mesleki gelişimlerinde yardımcı olmak ve onları desteklemek.

IEEE Yönetim Kurulu tarafından Ağustos 1990'da onaylanmıştır.

ÖNSÖZ

Proje konusu olarak Derin kapsül ağları ile yüz tanıma çalışması seçilmesindeki amaç Mimari tasarım stratejileri tarafından desteklenen derin evrişimli sinir ağları, nesne dönüşümlerini gömmek için çok sayıda özellik haritasına sahip veri büyütme tekniklerini ve katmanlarını kapsamlı bir şekilde kullanır. Bu, son derece verimsizdir ve büyük veri kümeleri için, özellik dedektörlerinin çok büyük miktarda fazlalığı anlamına gelir. Kapsül ağlar henüz başlangıç aşamasında olsalar da, mevcut evrişimsel ağları genişletmek ve yapay görsel algıya tüm özellik afin dönüşümlerini daha verimli bir şekilde kodlamak için bir süreç kazandırmak için umut verici bir çözüm oluşturuyor. Projemizdeki amaç iki farklı kapsül ağ modeli ile yüz veri seti üzerinde gerekli başarıya ulaşarak yüz tanımlamayı gerçekleştirmektir. Kısacası veri setinde en gelişmiş sonuçlara ulaşabilen kapsüllere dayalı verimli, yüksek oranda tekrarlanabilir, derin öğrenme sinir ağının kavramsallaştırılması ve geliştirilmesini hedeflemekteyiz. Bu çalışmamızda bizi dinleyip yönlendiren ve zamanını ayıran değerli Danışman Hocamız Murat AYKUT 'a ve Arş. Gör. Orhan SİVAZ' a teşekkürlerimizi sunarız.

Adı SOYADI
EKREM AĞCA
FEYZANUR MEMİŞ
KADER NUR TEKİN
MERVE AYDIN
Trabzon 2022

İÇİNDEKİLER

	Sayfa No
IEEE ETİK KURALLARI.....	II
ÖNSÖZ.....	III
İÇİNDEKİLER.....	IV
ÖZET.....	V
1. GENEL BİLGİLER.....	1
1.1.Giriş.....	1
1.2.KapsülAğları	4
1.3.EfficientNet	9
1.4.EfficientCapsnet:.....	13
1.5.DenseCapsuleNetworks.....	18
1.6.VeriSeti:YALE.....	25
2.YAPILAN ÇALIŞMALAR.....	26
2.1.OrjinalCapsnetin Yale VeriSeti Üzerinde Uygulanması	26
2.2.EfficientCapsnetin Yale VeriSeti Üzerinde Uygulanması	38
2.3.Dense Capsule Network'ün Yale Veri Seti Üzerinde Uygulanması	51
3. SONUÇLAR.....	65
4. KAYNAKLAR.....	66
STANDARTLAR ve KISITLAR FORMU.....	67

ÖZET

YALE VERİ SETİ İLE GELİŞMİŞ SONUÇLARA ULAŞABİLEN KAPSÜLLERE DAYALI VERİMLİ, YÜKSEK ORANDA TEKRARLANABİLİR VERİMLİ DERİN ÖĞRENME SİNİR AĞININ KAVRAMSALLAŞTIRILMASI VE GELİŞTİRİLMESİ

Yüz verilerini sınıflandırma işlemi, üzerinde çok sayıda çalışma yapılan derin öğrenme konularından birisidir. Bu konuda evrişimli sinir ağları, kolay uygulanabilir olmaları ve başarılı sonuçlar vermeleri nedenlerinden ötürü tercih edilen derin öğrenme uygulamalarının başında gelmektedir. Buna karşın evrişimli sinir ağlarında bulunan havuzlama katmanı verilerde bilgi kaybına neden olmaktadır. Ayrıca evrişimli sinir ağları, verideki bileşenlerin birbirine göre durumlarını göz ardı ederek eğitim işlemini gerçekleştirmektedir. Bu duruma çözüm olarak kapsül ağları önerilen derin öğrenme yöntemlerindendir. Bu çalışmada; 15 kişiden oluşan GIF formatında 165 gri tonlamalı görüntü içerir. Konu başına 11 görüntü, farklı yüz ifadesi veya konfigürasyonu başına bir tane vardır: merkez ışık, gözlüklü, mutlu, sol ışıklı, gözlüksüz, normal, sağ ışık, üzgün, uykulu, şaşırmış ve göz kırpma. Boyut: Veri kümesinin boyutu 6.4 MB'dir ve her biri 576 görüntüleme koşulunda görülen 10 nesnenin 5760 tek ışık kaynağı görüntüsünü içerir bu veri setini kullanmayı hedeflemekteyiz. Yapılan çalışmanın sonucunda kapsül ağlar yönteminin yale veri seti üzerindeki başarısını test etmeyi hedeflemekteyiz.

Karadeniz Teknik Üniversitesi
Bilgisayar Mühendisliği Anabilim Dalı
Danışman: Dr. Öğr. Üyesi Murat AYKUT

2022, 68 Sayfa

1. GENEL BİLGİLER

1.1 Giriş

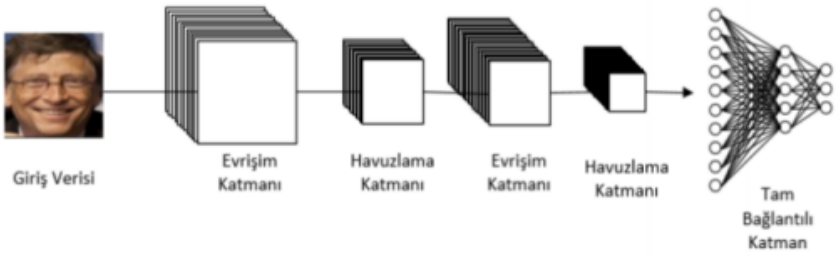
Bir grup insanın yüz görüntülerinden elde edilen verilerin belirlenen sınıflara göre sınıflandırılması yüz verilerinin sınıflandırılması işlemidir. Yüz verilerinin sınıflandırma işlemi günümüzde çok popüler olan evrişimli sinir ağları kullanılarak yapılmaktadır. Bizler bu çalışmamızda öncelikli olarak evrişimli sinir ağından bahsedip Yeni emekleme döneminde olan kapsül yapılarıyla bu sınıflandırma işlemini gerçekleştirmeye çalışacağız. Evrişimli sinir ağına geçmeden önce Derin öğrenme kavramından bahsedecek olursak;

Derin öğrenme bir makine öğrenme yöntemlerinden biridir. Elimizde bulunan bir veri kümesi ile çıktıları tahmin edecek yapay zekayı eğitmemize imkân sağlar. Kısacası yapay sinir ağlarının ve insan beyninden ilham alan algoritmaların veriden öğrendiği bir makine öğreniminin bir alt kümesidir. Derin öğrenme algoritması sonucu iyileştirmek amacıyla her defasında biraz değişiklik yaparak daha iyi iş çıkarmaktadır. Kullanılan en önemli alanlarından bir tanesi Yüz tanımadır.

Yüz tanımayı sadece güvenlik için değil aynı zamanda kullandığımız uygulamalar üzerinde kişileri etiketlemek amacıyla da kullanmaktayız. Örneğin facebook üzerinden gönderi paylaştığımız zaman insanları etiketlemek. Fakat burada karşılaştığımız en büyük sorun kişilerin saç, sakal vs. gibi şeylerinde stil değişikliğinde bulunduklarında veya alınan görüntüdeki kötü aydınlatma vs. gibi durumlarda bile aynı kişinin yüz tanımlamasını yapabilmeleridir.

Bu sorunları çözebilmek adına evrişimli sinir ağları geliştirilmiş ardından istenilen başarıya ulaşılamadığı için Kapsül ağları ortaya çıkmaktadır. Öncelikle Evrişimli sinir ağlarını inceleyelim.

Evrişimli sinir ağının ilk birkaç aşaması Evrişim(Convulation), Havuzlama(Pooling) katmanlarından oluşmaktadır. Son aşamada ise Tam Bağlı katman vardır ve Sınıflandırma katmanı ile yapı tamamlanmaktadır.
Evrişim katmanı kısacası ardışık birkaç eğitilebilir bölümlerden ve eğitici bir sınıflandırıcıdan oluşmaktadır.



Şekil 1: Evrişimli sinir ağı katmanları.
(Ayşe çoban, Fatih Özyurt, 2022).

Evrişim katmanı, giriş olarak alınan görüntülerin belirlenen boyutlarda ve sayılarda filtrelerden geçirildiği katmandır. Küçük boyutlu filtrelerin tüm görüntü üzerinde gezdirilmesine dayanmaktadır. Filtreler bir önceki katmandan gelen görüntü piksellerine konvolüsyon işlemi uygulayarak, her bir filtreye özgü özelliklerin keşfedildiği bölgeleri içeren aktivasyon(özellik)

haritasını oluşturmaktadır. Bu katman sayesinde görüntünün belirli sayıda filtreden geçirilmiş formları elde edilir. Bu katman bir veya birden fazla kez art arda uygulanabilir.

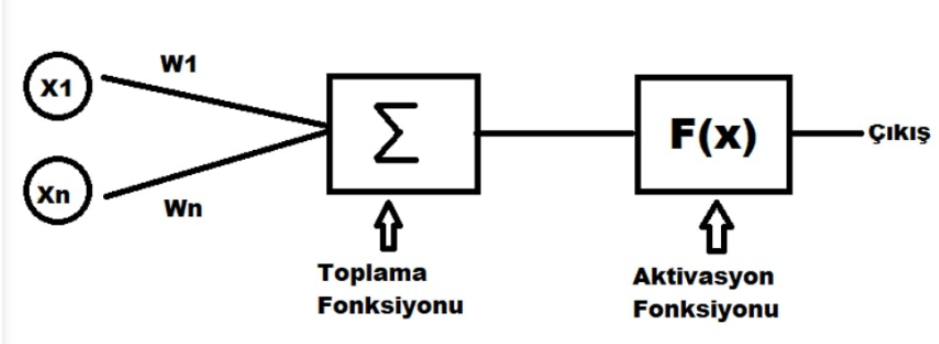
Evrişimli sinir ağlarının mimarisinde evrişim katmanından sonra havuzlama katmanı gelmektedir. Bu katmanın temel amacı bir sonraki konvolüsyon katmanı için giriş boyutunun azaltılmasıdır. Boyut azaltma işlemlerinde doğal olarak bazı bilgilerin kaybolması ve bundan dolayı performansın düşmesi söz konusudur.
Bu katman sonucunda havuzlama katmanının parametrelerine göre boyutları azaltılan ve istenilen öz niteliklere sahip görüntüler elde edilir.
Bu katman da modelin yapısına göre bir veya birden fazla kez tekrarlanabilir.

Son olarak tam bağlantılı katman gelmektedir. Bu katman basit anlamda bir yapay sinir ağıdır ve elde edilen görüntü matrisleri bir düzleştirme işleminden geçerek bu katmana iletilir. Bu katman kendinden önceki katmanın tün nöronlarıyla bağlantılı olup sınıf skorlarını optimize etmek için kullanılmaktadır. Bu katman sonunda bazı doğrusal olmayan aktivasyon fonksiyonları aracılığıyla sınıflandırma işlemi gerçekleştirilir.

Sınıflandırma katmanının çıkış değeri sınıflandırılması istenen nesne sayısına eşittir. Bu katmanda yaygın olarak softmax sınıflandırıcısı kullanılır. Sınıflandırma sonuncunda her sınıf için 0-1 aralığımda bir çıkış değeri üretilir. En yüksek olasılık değeri modelin tahmin ettiği sınıf olarak değerlendirilir.

Evrişim sinir ağına verilerin farklı perspektiflerden örneklerini verdiğimiz zaman her örnek için aynı başarı oranını vermektedir ve bu durum başlı başına sorun teşkil etmektedir. Bu sorunun en önemli nedeni havuzlama katmanında verilerde bilgi kaybının olmasıdır. Bu kayıp yüz verilerinin sınıflandırılmasında, sınıflandırmada etken olacak bazı öz niteliklerin kaybının olduğu anlamına gelmektedir.
Bu sorunun çözümü olarak kapsül ağları tercih edilen yöntemlerden biri olmuştur.

Öncelikle aktivasyon fonksiyonları nedir neden kullanılır bundan bahsedelim. Yapay sinir ağlarında doğrusal olmayan gerçek dünya özelliklerini tanıtmak için aktivasyon fonksiyonuna ihtiyaç duyuyoruz.



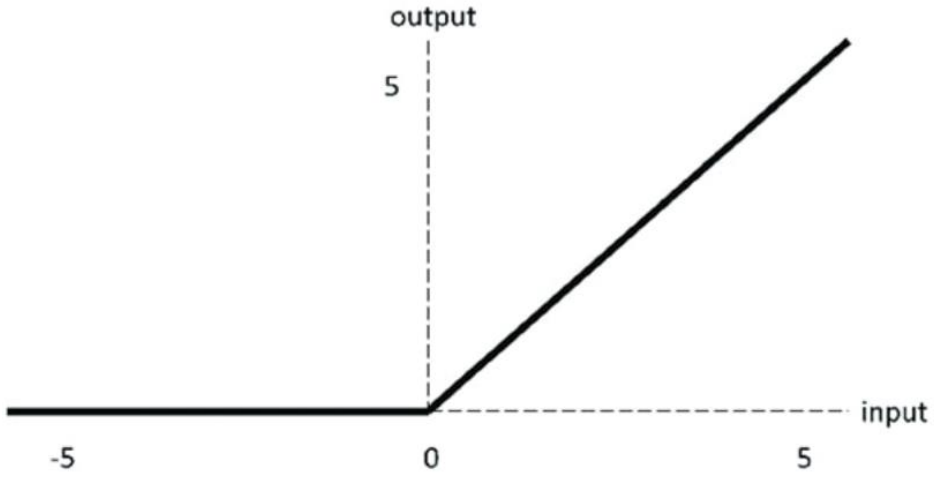
Şekil 2: Nöral Ağ Blok Diyagramı ve Aktivasyon Fonksiyonu

Şekil 3 'teki diyagram yapay sinir ağlarının temel yapı birimini oluşturmaktadır. Burada X1...Xn değerleri girdi olarak adlandırılırken W1...Wn ise bu girdilere karşılık gelen ağırlıkları temsil eder. Girdiler çeşitli ağırlıklar ile çarpılıp şekilde gösterilen toplama fonksiyonu ile

toplanır. Elde edilen değerler sistemde kullanılan aktivasyon fonksiyonundan geçirilerek çıkışa ulaşılır. Aktivasyonun buradaki görevi giriş sinyallerini çıkış sinyallerine çevirir.
Sinir ağlarında aktivasyon fonksiyonu uygulanmazsa çıkış sinyali basit bir doğrusal fonksiyon olmaktadır. Dolayısıyla aktivasyon fonksiyonu kullanılmayan bir sinir ağı sınırlı öğrenme gücüne sahip bir doğrusal regresyon gibi davranır. Gerçek dünyanın problemlerinin öğrenilmesi amaçlandığından verilerin değiştirilmesine ihtiyaç duyulmaktadır burada da devreye aktivasyon fonksiyonları girer.

ReLU (Rectified Linear Unit) Fonksiyonu

Derin sinir ağlarına dayalı olarak geliştirilen modellerde en yaygın kullanılan aktivasyon fonksiyonudur. Bu fonksiyon etkisiyle giriş verisindeki negatif değerler sıfıra çekilmektedir. Bunun sonucunda ağın daha hızlı öğrenilmesi sağlanmaktadır. Çıktı değerleri [0, +∞] arasında yer almaktadır.



Şekil 3: ReLU Aktivasyon Fonksiyonu

ReLU aktivasyon fonksiyonunun en fazla kullanılmasının nedeni 0'dan büyük olan girdilerin sabit türeve sahip olmasıdır. Böylelikle ağ daha hızlı eğitilebilir.

$$F(x)=MAX(X,0) \qquad (1)$$

ReLU fonksiyonunun tanımı denklem (1)'de gösterildiği gibidir. Burada fonksiyon çıktısı olan X, sıfırdan büyük değerler alınca sonuç X olurken, sıfırdan küçük değerler veya eşit değerler aldığında sonuç 0 olarak alınmaktadır. Böylelikle sadece pozitif değerlerde aktif olan ReLU aktivasyon fonksiyonları, sinir ağında ara katmanlardaki olası bir negatif çıkışı aktif etmeyecektir. Böylelikle sistem daha hızlı ve verimli çalışır.

Kapsül ağlarında kullanacağımız Squash Fonksiyonundan da bahsedecek olursak; Kapsül ağlarda ReLU fonksiyonuna karşılık Squash (ezme) fonksiyonu denilen bir aktivasyon fonksiyonu kullanılır. Evrişimli sinir ağlarında kullandığımız ReLU fonksiyonunun çıkışları skaler olarak ifade edilirken Squash fonksiyonunun çıkışları vektörel olarak ifade edilir. Vektörler, alınan veri bileşeninin hedef sınıfta bulunma olasılığını ve parametrelerini ifade etmektedir.Görüntü üzerinde istenilen nesnenin olduğu alanlarda vektör uzunluğu büyük olmadığı alanlarda ise küçüktür. Squash fonksiyonu vektörlerin uzunluğuna göre vektör uzunluklarını 0 ile 1 arasında bir değer almalarını sağlar.

Ezme(squash) fonksiyonu:

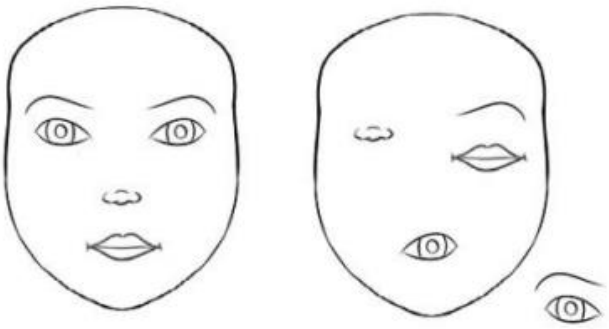
$$\mathbf{v}_j = \frac{||\mathbf{s}_j||^2}{1 + ||\mathbf{s}_j||^2} \frac{\mathbf{s}_j}{||\mathbf{s}_j||} \qquad (2)$$

Derin sinir ağlarında aktivasyon fonksiyonu olarak ReLU fonksiyonu kullanılırken kapsül ağlarında girdinin vektör olduğu squash fonksiyonu kullanılır.
Squash fonksiyonu sayesinde kapsül çıkışı, giriş vektörü kısa ise 0, uzun ise 1 şeklinde belirlenmiş olur

1.2 Kapsül Ağları

Kapsül ağlar verilerdeki bileşenlerin birbirine göre konum, açı, yönelim gibi özellikleri göz önüne alınarak eğitim işlemi yapılan ağlardır. Evrişimli sinir ağı görüntüdeki nesnelerin birbirlerine olan durumlarını, açısını, derinliğini tanımlamak için elverişli bir yöntem değil. Bu durum kullanılan havuzlama katmanlarından kaynaklanmaktadır. Havuzlama katmanında görüntünün boyutları düşürülürken görüntüdeki bazı öz nitelik kayıplarına neden olur. Kapsül ağları evrişimli sinir ağlarının yetersiz kaldığı problemler için önerilen yöntemdir.
Peki bu problem nedir?
Evrişimli sinir ağları nesne tanımlamada oldukça başarı göstermiştir. Örneğin bir yüz tanımlamada yüzü tespit eder fakat yüz üzerinde bulunan ağız ve burnun yer değiştirmesi durumunda bile aynı çıkış değerini üretmektedir. Buda es geçilemez bir sorundur. Bu durumda devreye kapsül ağları girmiştir. Öncelikle Kapsül nedir?
Birkaç temel özelliğe sahip küçük bir nöron grubudur. Bir kapsüldeki her nöron, belirli bir görüntü parçasının çeşitli özelliklerini temsil eder. Her kapsül, bir büyüklüğü ve yönü olan bir vektör verir ve bir kapsül ağı birden fazla kapsül katmanından meydana gelmektedir. Kapsül denmesinin sebebi, derinliğin artık katmanların art arda bağlamasıyla değil de iç içe kapsüllerle bağlanmasıyla elde edilmesinden dolayıdır.

Eğitim sırasında bu ağ bir nesnenin parçaları ve bütünü arasındaki ilişkileri öğrenmeyi amaçlar. Mesela, gözlerin ve ağzın tüm yüzün konumuyla nasıl ilişkili olduğu.
Kısacası yüzde bir yerlerde burun ve ağız var ama olması gereken yerde ve yönelimde mi evrişimli sinir ağı bunun kararını veremiyor, Bunların konumlarını göz önüne almadan eğitilmektedir ve burada devreye Kapsül ağları giriyor.

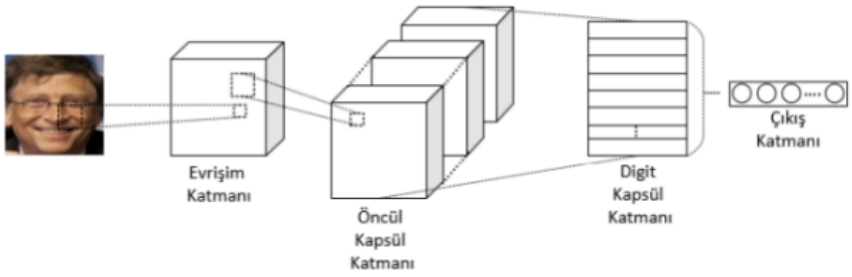


Şekil 4: Yüz bileşenlerinin birbirine göre durumları
(Onur Akköse, 2020).

.

Kapsül Ağlarının Mimari Yapısı

Evrişimli sinir ağlarındaki nöronların yerine kapsül ağlarda kapsüller kullanılır. Kapsül ağı katmanları temel olarak Şekil 5'te gösterildiği gibidir.



Şekil 5: Kapsül ağı katmanları
(Ayşe çoban, Fatih Özyurt, 2022).

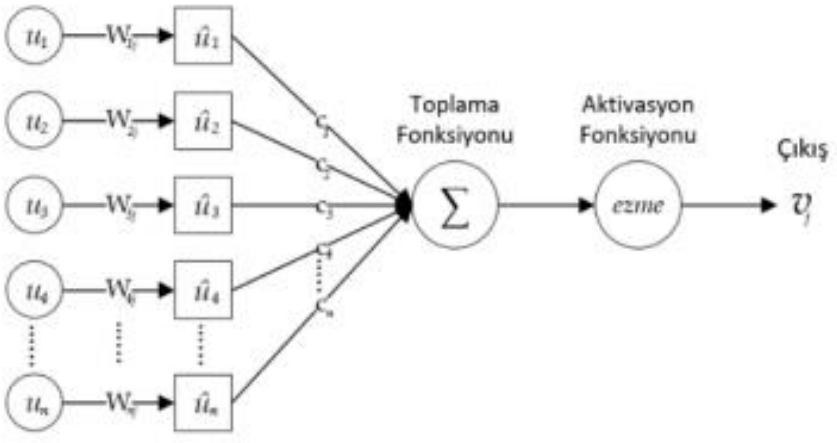
Kapsül ağının ilk aşamasında kullanılan evrişim katmanı evrişimli sinir ağındakiyle aynıdır. Fakat bu evrişim katmanından sonra gelen evrişim katmanında ve diğer katmanlarda dinamik yönlendirme algoritması uygulanmaktadır.
Belirli bir giriş görüntüsünde kenarlar gibi özellikleri çıkarmayı öğrenecek olan katmandır. Çıkışına özellik haritası oluşturmak adına RELU fonksiyonu uygulanır.

İkinci katmanda ise öncül(primary) adı verilen bir katman yer almaktadır. Bu katmanda yine bir evrişim işlemi yapılır. Uygulanan evrişim işleminin bir önceki evrişimden farkı bu katmana gelen görüntüleri vektörel forma getirmek için yeniden boyutlandırma işlemi uygulanması ve ardından aktivasyon fonksiyonu olarak squash fonksiyonunun kullanılmasıdır. Bu işlemler sonucunda dinamik yönlendirme algoritmasının kullanıldığı katman bulunmaktadır. Bu katman sonucunda DigitCaps olarak adlandırılan vektörler elde edilmektedir. Bu vektörlerin sayısı sınıflandırma işlemi için belirlenen sınıf sayısı kadardır.

Dinamik Yönlendirme Algoritması

Dinamik yönlendirme algoritması, girişleri benzer olan alt seviyede olan kapsülleri, girişleri benzer olan üst seviyedeki kapsüllerle eşleştiren bir algoritmadır. Kısacası bir sonraki kapsül çıkışını hesaplamak için kullanılır. Kapsül ağlarında evrişimli sinir ağlarında kullanılan ReLU fonksiyonu yerine ezme(squash) fonksiyonu kullanılır. Squash fonksiyonunun çıkışı vektörel olarak ifade edilir. Buradaki vektörler veri bileşeninin hedef sınıfta bulunma olasılığını ve parametrelerini ifade eder.

Dinamik yönlendirme algoritmasının kullanılması havuzlama yöntemine göre öznitelik çıkarmada daha etkili bir yöntemdir.



Şekil 6: Kapsül giriş ve çıkışları.
(Ayşe çoban, Fatih Özyurt, 2022).

u_i kapsül girişlerini ifade ederken W_ij ise bu vektörlere uygulanan dönüşüm matrislerini ifade etmektedir. Sonuç olarak tahmin vektörü olan u_ij elde edilir.

Daha sonra tahmin vektörünün, kapsül i'nin kapsül j'ye kadar aktive edebileceğinin ölçümü olan c_ij ile çarpımlarının toplamı alınır bu da ağırlıklı bir toplam olan s_j'yi verir.

$$s_j = \sum_i c_{ij} \hat{u}_{i|j} \tag{3}$$

$$\hat{u}_{i|j} = W_{ij} u_i \tag{4}$$

Hesaplamış olduğumuz bu değerlere ezme fonksiyonu uygulayarak v_j'yi elde ederiz. (Denklem 5).

$$v_j = \frac{\|s_j\|^2}{1 + \|s_j\|^2} \frac{s_j}{\|s_j\|} \tag{5}$$

Benzerlik ise tahmin vektörü u_ij ile kapsül çıkışı olan v_j'nin çarpımlarıyla hesaplanır.

$$b_{i|j} \leftarrow \hat{u}_{i|j} v_{ij} \tag{6}$$

Hesaplamış olduğumuz b_ij 'ye softmax uygulanması ile c_ij'nin yeni değeri elde edilmiş olacaktır, b_ij yi daha doğru hale getirmek için yinelemeli olarak güncellenir.

$$c_{ij} = \frac{\exp(b_{ij})}{\sum_k \exp(b_{ik})} \tag{7}$$

$$b_{i|j} \leftarrow b_{i|j} + \hat{u}_{i|j} v_{ij} \tag{8}$$

Eğitim başlangıcında b_ij değeri sıfırdan başlatılır.

---

1. **algoritma** Yönlendirme $(\hat{u}_{i|j}, r, l)$
2. tüm kapsüller i, katmanlar l ve bulunulan kapsül katmanı j olmak üzere $(l+1)$: $b_{ij} \leftarrow 0$
3. **for** r iterasyon **do**
4. tüm kapsüller için i. katman l: $c_i \leftarrow softmax(b_i)$
5. tüm kapsüller için j. katman $(l+1)$: $s_j \leftarrow \sum_i c_{ij} \hat{u}_{i|j}$
6. tüm kapsüller için j. katman $(l+1)$: $v_j \leftarrow ezme(s_j)$
7. tüm kapsüller için i. katman ve l kapsül j. katman $(l+1)$: $b_{ij} \leftarrow b_{ij} + u_{j|i} v_j$

return v_j

Şekil 7: Dinamik Yönlendirme Algoritması

Kısaca şekil 7'yi anlatacak olursak;
İlk satırda r, algoritmayı yenileme sayısını, l seviyesindeki tüm kapsülleri ve bunların çıktısı olan (u_ij) aldığını ifade eder.
En son satırda ise algoritmanın sonucunda daha yüksek seviyeli bir kapsül çıktısı olan v_ij üretileceğini ifade eder.
b_ij katsayısı yinelemeli olarak güncellenen bir değerdir ve prosedür bittikten sonra değeri c_ij 'de saklanır.
4.satırda ise daha düşük seviyeli bir kapsül i için tüm yönlendirme ağırlıkları olan c_i vektörünün değeri hesaplanır. Bu sırada b_i ise değerlerini softmax fonksiyonundan geçirir. Tüm düşük seviyeli kapsüller için bu işlem gerçekleştirilir. İlk yinelemede, tüm c_ij değerleri eşit olacaktır, nedeni ise ilk etapta bütün b_ij değerleri sıfıra ayarlanmıştır. Yinelemeler oldukça bu tek tip dağılımlar değişecektir.

Düşük seviyede bulunan tüm kapsüller için c_ij yani tüm ağırlıklar hesaplandıktan sonra, daha yüksek kapsüllere bakılması için 5.satıra geçilir.
Burada önceki adımlarda belirlenen c_ij yönlendirme katsayıları ile ağırlıklandırılan girdi vektörlerinin doğrusal bir kombinasyonu hesaplanır böylelikle s_j çıktı vektörünü üretmiş oluruz.
Yüksek seviyeli tüm kapsüller için bu işlem uygulanır. Son durumdaki vektörler vektör yönünün korunmasını sağlayan ve uzunluğunun 1' den fazla olmamasını sağlayan squash fonksiyonundan geçirilir. Bu durumda tüm yüksek seviyedeki kapsüller için v_j çıktı vektörünü üretilmiş olur.
Bu adımlar her bir yüksek seviyeli j'ye bakar ve daha sonra her girişi inceler ve formüle karşılık gelen b_ij ağırlıklarını günceller. Böylelikle düşük seviyeli kapsül, çıktısını benzer olan daha yüksek seviyeli kapsüle göndermiş olur.
Bu algoritma yinelemelere 3. adımdan başar ve r kez tekrar eder. Bunun sonucunda evrişimli sinir ağlarından farklı olarak nesnelerin birbirine göre durumlarını da hesaba katmış oluyor.

Özetle:
Dinamik yönlendirme bir kapsül katmanının çıktısıyla sonraki kapsül katmanının girdileri arasındaki iletişimin kurulmasını sağlar.
Bir kapsül ağı ne tür bir girdi görürse görsün dinamik yönlendirme primary katmanındaki bir alt kapsülün çıktısının digit katmanındaki an alakalı üst kapsüle gönderilmesini sağlar.

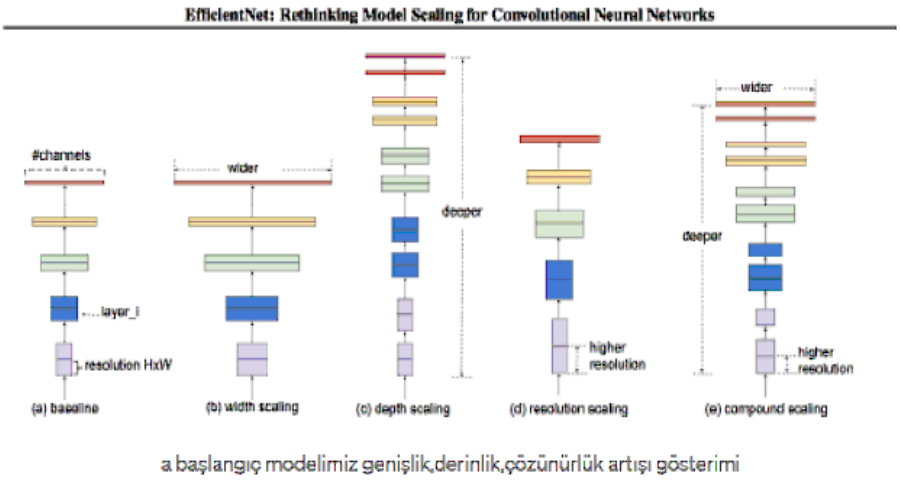
1.3 EfficientNet

Bir grup evrişimli sinir ağı modeli olarak görülebilir. Transer öğrenimi olarak da isimlendirilir. Kısaca bir görev için eğitilmiş modelin, ilgili ikinci bir görevde yeniden tasarlandığı bir makine öğrenmesi tekniğidir. Daha az veri ve işlem gücü kullanrak performans artışı sağlayabiliriz. Örnek verecek olursak, ben bir bardağı tanımak için atalarımdan bu yana ve doğduğumdan beri öğrendiğim basit özellikler var (kenar, köşe, şekil, maddesel yapısı vb.) bunlardan yola çıkarak hiç görmediğim bardakları ya da hiç görmediğim bazı nesnelerin bardak olmadığına dair kararlar veriyorum. Yalnızca bu bilgiyi öğrenen bir makinenin bildiklerini başka bir makineye transfer edip tekrar öğrenme sürecini atlamasıdır diyebiliriz.

B0-B7 arasında 8 tane modelden oluşuyor ve sayı büyüdükçe parametre sayısı ve doğrulukta artar.
Bu modelde 3 aşamada ölçeklendirme yapmak gerekiyor bunlar derinlik, genişlik ve çözünürlüktür.

Derinlik parametresi: Ağların içerisindeki katman sayısına denk gelir. Daha derin ağlar daha zengin ve karmaşık özellikleri yakalayabilir.

Genişlik parametresi: Bir katmanımızdaki nöron sayısı olarak düşünebiliriz. Daha geniş ağlar ince ayrıntıları daha çok yakalayabilirler.
Çözünürlük parametresi: Modelimizde eğiteceğimiz resimlerin çözünürlüğü diyebiliriz.

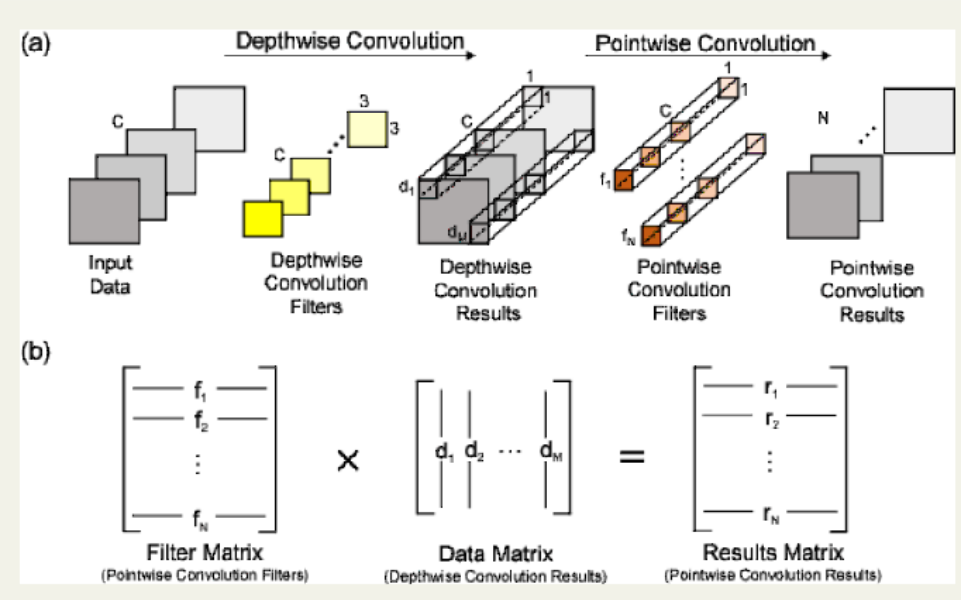


Şekil 8: Farklı ölçekleme yöntemleri ve Bileşik ölçekleme.
(Mingxing Tan 1 Quoc V. Le 1,2020).

Model Ölçeklendirme. (a) bir temel ağ örneğidir; (b)-(d) ağın sadece bir boyutunu artıran geleneksel ölçeklendirmedir: genişlik, derinlik veya çözünürlük. (e) her üç boyutu da sabit bir oranla eşit şekilde ölçeklendiren önerilen bileşik ölçeklendirme yöntemidir.

Tek boyutları ölçeklendirmenin model performansını iyileştirmeye yardımcı olurken, ölçeği üç boyutun tamamında (genişlik, derinlik ve görüntü çözünürlüğü) dengelemenin, mevcut değişken kaynakları göz önünde bulundurarak genel model performansını en iyi şekilde iyileştirdiği sonucuna bu yöntemle varıldı.

1.3.1 EfficientNet Nasıl Çalışır?



Şekil 9: Depth ve Point wise Evrişimlerin temel gösterimi.
(Ayyüce Kızrak , 2019).

1.Depthwise Evrişim + Pointwise Evrişim: hesaplama maliyetini önemli ölçüde azaltmak için orijinal evrişimi iki aşamaya böler.
2.Ters çevrilmiş Res: orjinal ResNet blokları Seqeeze ve expand kanallarından oluşur.
Böylece atlama bağlantıları ile zengin kanal bağlantılarını bağlar fakat MBConv 'da bloklar ilk başta kanalları genişleten daha sonra sıkıştıran bir katmandan oluşur böylece daha az kanala sahip katmanlar atlanarak bağlanır.
3.Lineer darboğaz: ReLU'dan bilgi kaybını önlemek için her bloktaki son katmandaki lineer aktivasyonu kullanılır.

Bileşik ölçekleme yöntemi, sabit bir oranla ölçeklendirme yoluyla genişlik, derinlik ve çözünürlük boyutlarının dengelenmesi fikrine dayanır. Aşağıdaki denklemler matematiksel olarak nasıl elde edildiğini gösterir,

$$\text{Derinlik (depth): } d = \alpha^{\varphi} \quad (9)$$

$$\text{Genişlik (width): } w = \beta^{\varphi} \quad (10)$$

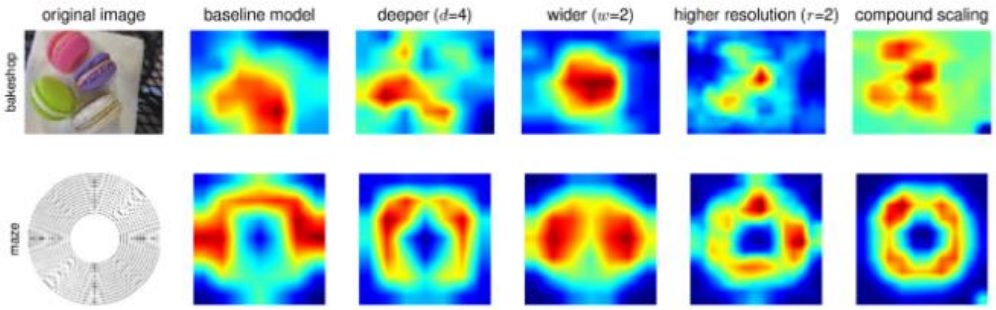
$$\text{Çözünürlük (resolution): } r = \gamma^{\varphi} \quad (11)$$

$$\text{s.t. } \alpha \cdot \beta^2 \cdot \gamma^2 \approx 2$$

$$\alpha \geq 1, \beta \geq 1, \gamma \geq 1$$

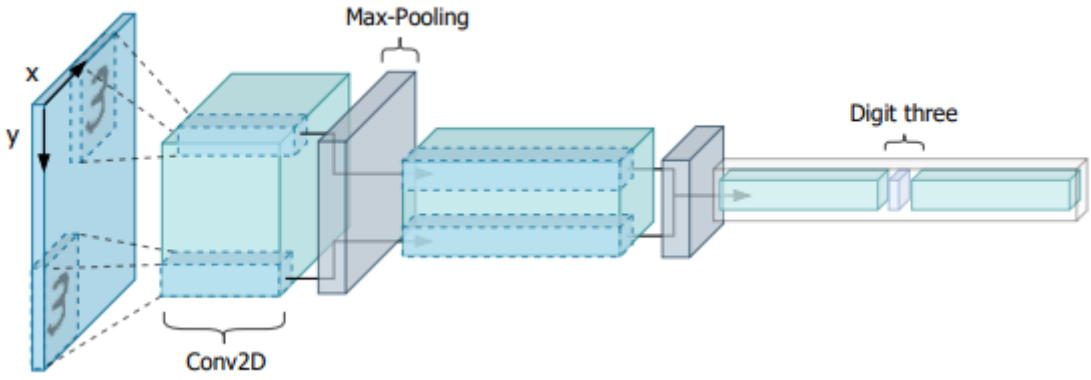
EfficientNet, modelleri basit ama etkili bir şekilde ölçeklendirmek için bileşik katsayı adı verilen bir teknik kullanır. Genişliği, derinliği veya çözünürlüğü rastgele ölçeklendirmek yerine, bileşik ölçekleme, belirli bir sabit ölçekleme katsayıları seti ile her bir boyutu eşit olarak ölçekler.

Şekil 10 ‘da görüldüğü gibi, modelin ayrıca daha fazla nesne detayıyla ilgili bölgelere odaklanan ve daha iyi model açıklanabilirliğine giden yolu açan daha iyi Sınıf Aktivasyon Haritaları (CAM) sağladığı görülmüştür.



Şekil 10: Bileşik ölçekleme yöntemi, ölçeklenen modelin (son sütun) daha fazla nesne detayıyla daha alakalı bölgelere odaklanmasını sağlar.
(Ayyüce Kızrak , 2019).

Evrişimli sinir ağları (CNN), yapay görsel algıyı büyük ölçüde değiştirerek, görüntü sınıflandırmadan [1, 2, 3] nesne algılamaya [4, 5, 6] ve örnek olaya kadar bilgisayarla görmenin tüm temel alanlarında dikkate değer sonuçlar elde ettiği gözlenmiştir segmentasyon [7]. CNN'nin temel özelliği, aynı bilgiyi bir girdi görüntüsünün uzamsal boyutundaki tüm konumlarda verimli bir şekilde çoğaltma yeteneğidir. Diğer derin sinir mimarilerinin aksine.
Öğrenilen özellik algılayıcılarının çevrilmiş kopyaları kullanılarak, bir uzamsal konumda(spatial location) öğrenilen özellikler diğer konumlarda mevcuttur. Maksimum havuzlama (Max-pooling) gibi uzamsal indirgeme katmanlarıyla birleştirilmiş yerel paylaşılan bağlantı, yerel çeviriyle değişmez özellikleri çıkarır. Bu nedenle, Şekil 11'de gösterildiği gibi, giriş alanındaki nesne çevirileri, yüksek seviyeli nöronların aktivasyonlarını etkilemez, çünkü maksimum havuzlama katmanları, katmanlar arasında düşük seviyeli özellikleri yönlendirebilir. CNN tarafından elde edilen çeviri değişmezliği(translation invariant), nesnelerin konumunun kesin kodlamasını kaybetme pahasına gelir. Ayrıca, CNN'ler diğer tüm yakın dönüşümlere de alternatif değildir.



Şekil 11: Uzamsal indirgeme için maksimum havuzlama katmanlarına ve düz bir uzaysal çeviri ile elde edilen iki girdi nesnesine sahip basit bir CNN'nin sıkıştırılmış temsili. Maksimum havuzlama işlemleri, ilkel yönlendirme rolleri her iki basamak için vurgulanacak şekilde şematize edilir. Ağın önceki aşamasında tespit edilen düşük seviyeli özellikler, kademeli olarak ortak yüksek seviyeli özelliklere yönlendirilir. Bu nedenle, model çeviri değişmezdir, ancak ilgili nesne yerelleştirme bilgilerini yavaş yavaş kaybeder (Vittorio Mazzia, Francesco Salvetti, Marcello Chiaberge, 2021).

Bu sorunu dengelemek için farklı teknikler geliştirildi. Benimsenen yaygın çözümlerin çoğu, daha fazla sayıda özellik eşlemesinin kullanılmasını, ağın tüm ek dönüştürmeler için yeterli özellik algılayıcıyla donatılmasını sağlayacak şekilde sağlar. Öğrenilecek farklı pozları oluşturmak için veri iyileştirme teknikleri kullanılır ve artık bağlantılar ve normalleştirme teknikleri ağ filtre kapasitesinin artırılmasına olanak tanır. Ancak tüm bu ilave mekanizmalar CNN'in iç sınırlamalarını kısmen telafi ederek modelin eğitim sırasında karşılaşılan aynı nesnelerin farklı dönüşümlerini tanımasını engeller. Gerçekten de büyük veri setleri konusunda eğitim almış CNN'ler, dedektörlerin çok fazla özelliğe sahip olması ve ilgili bakış açılarıyla binlerce nesneye ölçeklendirmede zorluk yaşamaktadır. Görüş noktası değişikliklerinin piksel uzayı üzerinde karmaşık etkilere sahip olması, ancak bir nesne ile tüm nesne arasındaki ilişkiyi temsil eden duruş üzerindeki basit doğrusal etkilerdir.
Genel bir tam bağlantılı veya kıvrımlı derin nöral ağda, belirli bir özelliğin varlığını göstermek amacıyla özellik dedektörlerinin ve nöron aktivasyonlarının kodlanmasında ağırlıkları kullanırız. Bu nedenle, eğitimden sonra ağırlıkları düzelten model, eğitim sırasında karşılaşılmayacak basit dönüşüm modellerini tespit edemiyor. Öte yandan, nesne özellikleri arasındaki ilişkileri gömmek(embed) için ağırlıkların yeniden kullanılmasını değerlendirmemiz gerekmektedir. Aslında, parçalar arasında yapısal bir dönüşüm ve bakış açısına göre tam bir varyant olarak, ağırlıklar bunları verimli bir şekilde temsil etmek için mükemmel bir şekilde uyarlanmıştır ve otomatik olarak yeni bakış açılarına sahip olmaları gerekir. Ayrıca, artık dönüşümler için farklı olmayan aktivasyonlar elde etmek istemiyoruz, aynı varlığın farklı özelliklerini temsil etmek için sinerji içinde çalışan nöron gruplarını elde etmeyi istiyoruz. Kapsüller, özelliklerin vektör gösterimleridir ve bakış açısı dönüşümüne eşdeğerdirler.

Bu nedenle, her kapsül yalnızca belirli bir obje tipini temsil etmekle kalmaz, aynı zamanda varlığın nasıl somutlaştırıldığını dinamik olarak tanımlar ve içgüdüsel olduğunu da dinamik olarak açıklar. Son olarak, öğrenilen özellik dedektörleriyle eşleşen puana dayalı olarak bir skaler birimin etkinleştirildiği geleneksel ağların çalışma prensibi, çok daha sağlam bir mekanizma tamamen bırakılmıştır. Aslında, ağırlıklarda kodlanmış bakış açısı varyant dönüşümleri ile, kapsüllerin bunların bir parçası olmaları gerektiğini tahmin edebiliriz.
Bu nedenle, yüksek seviyeli kapsülleri etkinleştirmek için düşük seviyeli kapsüllere uygun tahminler yapmayı düşünebiliriz. Anlaşmaları ölçmek için bir süreç gerektirir ve kapsülleri en iyi eşleşen ana öğeye yönlendirin. İlk olarak, dinamik yönlendirme, anlaşmaya göre ilk yönlendirme mekanizması olarak önerildi. Tahminlerde bulunmak ve karşılıklı anlaşmalarını değerlendirmek için nöron aktivasyon gruplardan yararlanma, kovaryansı yakalamanın çok daha etkili bir yoludur ve önemli ölçüde azaltılmış sayıda parametreye ve çok daha iyi genelleme özelliklerine sahip modellere yol açmalıdır. Bununla birlikte, kapsül ağlarının verimlilik açısından ve bilgi nesnesi dönüşümlerini daha iyi temsil etme konusundaki içsel yeteneklerine çok az ilgi gösterilmiştir. Aslında, şimdiye kadar sunulan tüm model çözümleri, kapsüllerin sağlaması gereken içsel genelleme özelliğini kaçınılmaz olarak gizleyecek çok sayıda parametreyi dikkate almaktadır. Bu yazıda, orijinal CapsNet modeline göre neredeyse

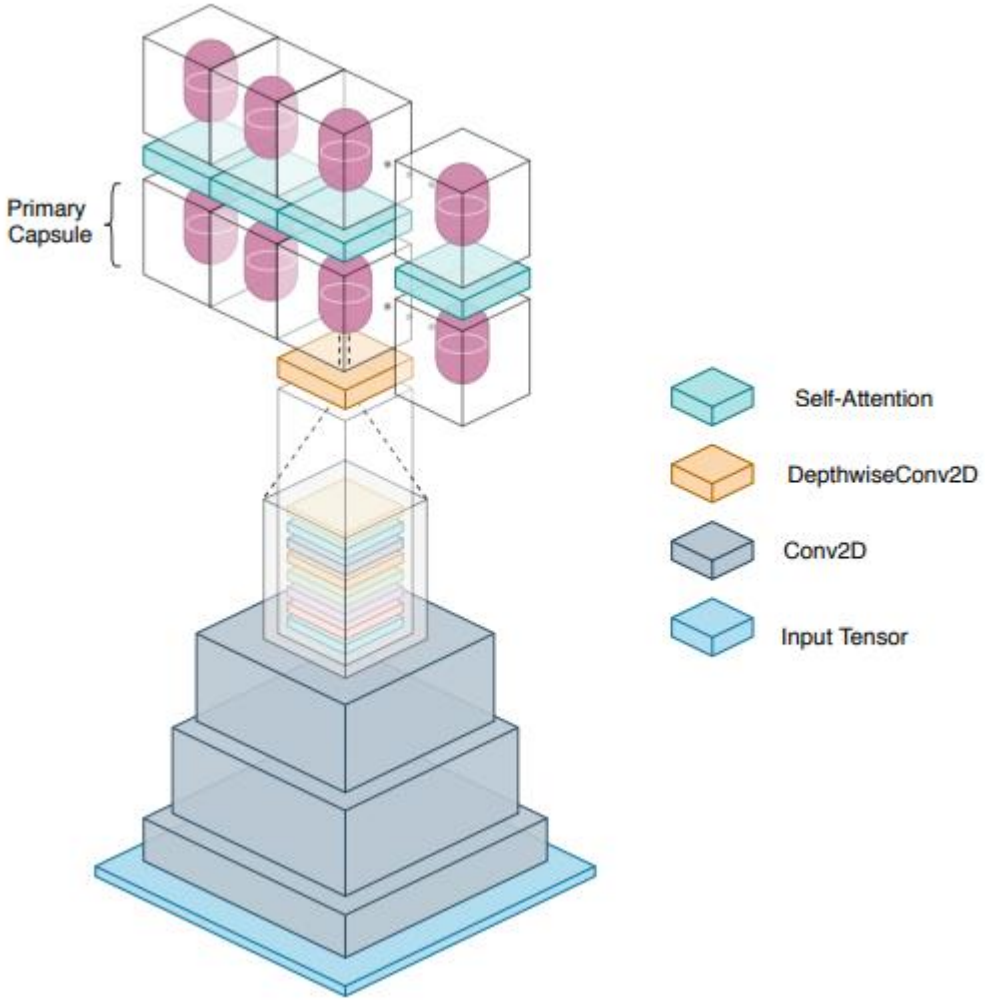
160K parametresi ve %85 TOPs iyileştirmesi ile uç bir mimari olan Efficient-CapsNet'i öneriyoruz.
Genel olarak, çalışmamızın ana katkısı şunlarda yatmaktadır:

• Kapsüllere dayalı ağların genelleme gücünün derinlemesine araştırılması, önceki literatür araştırma çalışmalarına kıyasla eğitilebilir parametrelerin sayısını büyük ölçüde azaltır.
• En gelişmiş sonuçlara ulaşabilen kapsüllere dayalı verimli, yüksek oranda tekrarlanabilir, derin öğrenme sinir ağının kavramsallaştırılması ve geliştirilmesi.

Yöntem

1.4 Efficient-CapsNet

Efficient -CapsNet'in genel mimarisi Şekil 12'de gösterilmiştir. Üst düzey bir tanım olarak ağ genel olarak üç farklı bölüme ayrılabilir; burada ilk ikisi, giriş alanı ile etkileşime girmek için birincil kapsül katmanının ana araçlarıdır. Aslında her kapsül, piksel yoğunluklarını etki yaptığı özelliğin vektörel gösterimlerine dönüştürmek için aşağıdaki kıvrımlı katman filtrelerinden faydalanan bir özelliktir. Bu nedenle, aktif bir kapsül içindeki nöronların faaliyetleri, eğitim sürecinde temsil etmeyi öğrendiği varlığın çeşitli özelliklerini bünyesinde barındırmaktadır.



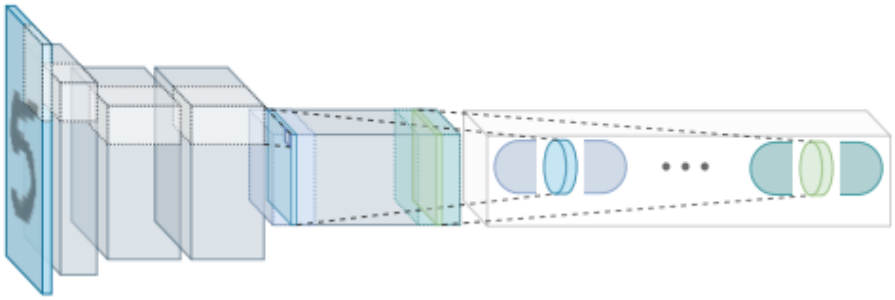
Şekil 12: Efficient-CapsNet'in genel mimarisinin şematik gösterimi. Birincil kapsüller, temsil ettikleri özelliklerin vektörel bir temsilini oluşturmak için derinlemesine ayrılabilir evrişim den yararlanır. Öte yandan, evrişim katmanlarının ilk yığını, giriş tensörünü daha yüksek boyutlu bir uzaya eşler ve kapsüllerin oluşturulmasını kolaylaştırır (Vittorio Mazzia, Francesco Salvetti, Marcello Chiaberge, 2021).

Özellikler; poz, doku, deformasyon ve unsurun kendisinin varlığı gibi birçok farklı türde örnekleme parametresi içerebilir. Uygulamamızda, her vektörün uzunluğu, bir kapsül tarafından temsil edilen varlığın mevcut olma olasılığını temsil etmek için kullanılır.
Bu, herhangi bir mantıklı amaç fonksiyonu minimizasyonu gerektirmeyen self-attention algoritması ile uyumludur. Ayrıca, mevcut olmayan objeleri temsil etmek için büyük faaliyetler kullanmadığı için biyolojik açıdan anlamlıdır. Son olarak, ağın son bölümü, düşük seviyeli kapsülleri temsil ettikleri bütüne yönlendirmek için self-attention algoritması altında çalışır.

Daha resmi olarak, tek bir örnek (i) durumunda model, H, W ve C'nin tek giriş görüntüsünün yükseklik, genişlik ve kanalları/özellikleri olduğu H × W × F şeklinde bir tensor X olarak temsil edilen bir görüntü girişi olarak kabul eder. Birincil CAPS katmanına girmeden önce, bir dizi hacimsel ve Batch Normalization (Toplu Normalleştirme) katmanı aracılığıyla giriş görüntüsünden X yerel özellikleri çıkarırız. Bir konvolüsyon katmanının her çıktısı l belirli bir çekirdek boyutu k, unsur haritalarının sayısı f, adım s=1 ve ReLU ile bir hacimsel işlemle oluşturulur.

$$\mathrm{F}^{l+1}(\mathbf{X}^l) = \mathrm{ReLU}\left(Conv_{k\times k}(\mathbf{X}^l)\right) \tag{12}$$

Genel olarak, ağın ilk evrişimsel kısmı, giriş görüntüsünü kapsül oluşturmayı kolaylaştıran daha yüksek boyutlu bir alana eşleyen tek bir işlev HConv olarak modellenebilir. Öte yandan ağın ikinci bölümü, birincil kapsüller tarafından özelliklerin vektörel bir sunumunu oluşturmak için birincil kapsüller tarafından kullanılan ana araçtır.



(13)

Şekil 13: Ağın ilk kısmı, giriş görüntüsünü daha yüksek boyutlu bir alana eşleyen tek işlevli HConv olarak modellenebilir. Ardından, birincil kapsül katmanı S^1n,d, kapsüllerin oluşturulması için gereken parametre sayısını büyük ölçüde azaltan, derinlemesine ayrılabilir bir evrişim ile elde edilir (Vittorio Mazzia, Francesco Salvetti, Marcello Chiaberge, 2021).

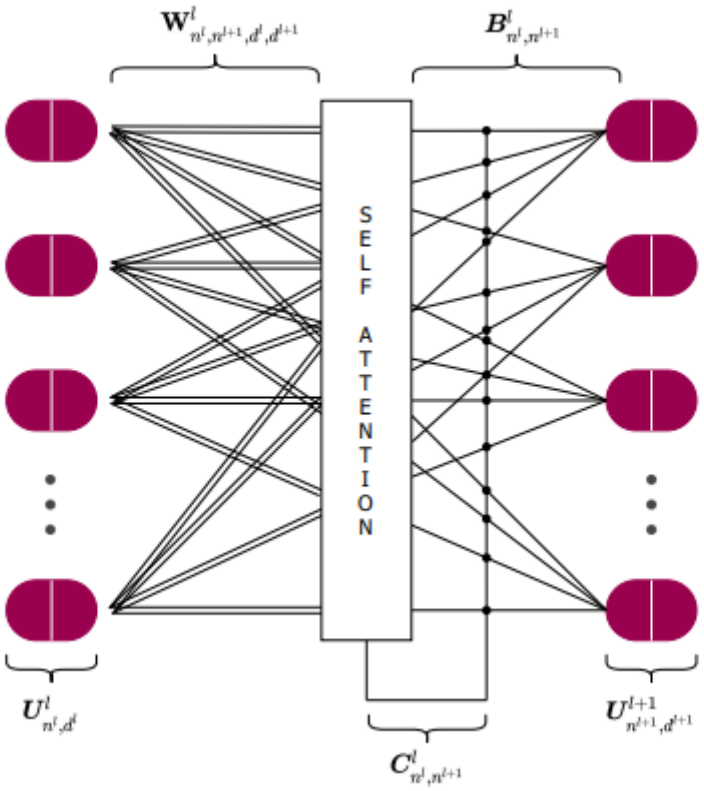
Şekil 13'te gösterildiği gibi, her kanalda ayrı olarak hareket eden, derinlik açısından uzaysal

konvolüsyon işleminin yalnızca ilk adımını gerçekleştiren doğrusal etkinleştirmeye sahip, bir evrişim işleminin sadece ilk adımını gerçekleştiren lineer aktivasyonlu derinlemesine ayrılabilir bir evrişimdir.

Bu işlemden sonra konum bilgisi artık kapsüllerin özelliklerinde "yer kodlu" (place-coded) değil "oran kodlu"(rate-coded) olur. Dolayısıyla, ağın temel elemanı artık tek bir nöron değil, vektör çıkışlı bir kapsüldür. Gerçekten de birincil kapsül tabakasına uygulanan ilk işlem, kapsül şeklinde bir aktivasyon fonksiyonudur. Vektörlerin uzunluğu ile belirli bir varlığın var olma olasılığını kodlamak ve aktif kapsüllerin daha yüksek seviyeli kapsüllerin örnekleme parametreleri için tahminler yapmasına izin vermek için, aktivasyon fonksiyonu tarafından iki önemli özelliğin karşılanması gerekir; bir vektör yönelimini korumalı ve sıfır ile bir arasındaki uzunluğu korumalıdır. Efficient-CapsNet, orijinal etkinleştirme işlevinin squash işlemi olarak adlandırılan bir türevini kullanır:

$$\text{squash}(\boldsymbol{s}_n^l) = \left(1 - \frac{1}{e^{||\boldsymbol{s}_n^l||}}\right) \frac{\boldsymbol{s}_n^l}{||\boldsymbol{s}_n^l||} \tag{13}$$

Denklemin squash fonksiyonu (13), gerekli iki özelliği karşılar ve sıfıra yakın küçük değişikliklere karşı çok daha hassastır, eğitim aşaması sırasında gradiyente bir artış sağlar. Aslında doğrusal olmama, kısa vektörlerin neredeyse sıfır uzunluğa, uzun vektörlerin ise birin biraz altında bir uzunluğa küçülmesini sağlar.



(14)

Şekil 14: l − th Katmanın kapsülleri, parçası olabilecekleri bütünün tahminlerini yapar. Ağırlık tensörü(W1) ile elde edilen tüm tahminler.(Vittorio Mazzia, Francesco Salvetti, Marcello Chiaberge, 2021).

Self-Attention routing

Aktif kapsülleri ait oldukları bütüne yönlendirmek için self-attention routing algoritmamızdan yararlanıyoruz. Şekil 14'te gösterildiği gibi, ek boyuta rağmen, genel mimari, öz-dikkat algoritmasının getirdiği ek bir dal ile tam bağlantılı bir ağa çok benzer. Gerçekten de yukarıdaki katmandaki bir kapsülün toplam girdisi, s^1+1n, aşağıdaki katmandaki u^1n kapsüllerinden gelen tüm "tahmin vektörleri" üzerinden ağırlıklı bir toplamdır. Bu, bir ağırlık matrisi için U ^1n, d'ye ait olan her bir kapsülün, u ^1 n matris çarpımı tarafından üretilir. Sezgisel olarak, tüm ağırlık matrislerini içeren W^1 n^1, n^1+1, d^1, d^1+1 tensörünün tamamı, iki bitişik katmanın kapsülü arasındaki tüm afin dönüşümleri gömer. Yani, katmanın her kapsülü, yukarıdaki tabaka için izdüşümlerini yapmak için. Eşitlik 14 takip edelim.

$$\hat{\mathbf{U}}^{l}_{(n^l,n^{l+1},:)} = \boldsymbol{u}^{\mathrm{T}l}_{n} \times \mathbf{W}^{l}_{(n^l,n^{l+1},:,:)} \quad (14)$$

Burada U^1n^1, n^1+1, d^1+1, l- inci kapsüllerin tüm tahminlerini içerir. Gerçekten de ağırlık matrisi aracılığıyla her n ^1 kapsül, tüm n ^1+1 kapsüllerin özelliklerini tahmin eder. Gerçekten de yukarıdaki katmanın kapsülleri, s ^1+1n, Denklem ile hesaplanabilir. (15)

$$\boldsymbol{s}_n^{l+1} = \hat{\mathbf{U}}^{\mathrm{T}l}_{(:,n^{l+1},:)} \times \left(\boldsymbol{C}^l_{(:,n^{l+1})} + \boldsymbol{B}^l_{(:,n^{l+1})} \right) \tag{15}$$

Burada B^1n^1, n^1+1, diğerleriyle aynı anda ayrımsal olarak öğrenilen tüm ağırlıkları içeren log öncelik matrisidir. Ağırlıklar öte yandan, C^1 n^1, n^1+1, self-attention algoritması tarafından üretilen tüm birleştirme katsayılarını içeren matristir. Bu nedenle, öncelikler daha bağlantılı kapsüllere yönelik önyargılar oluşturmaya yardımcı olur ve self-attention routing, algılanan şekilleri dikkate alınan belirli (i) örnekte temsil ettikleri bütüne dinamik olarak atar. Bağlantı katsayıları, self-attention tensörü A^1 n^1, n^1, n^1+1'den başlayarak Denklem kullanılarak hesaplanır. (16)

$$\mathbf{A}^l_{(:,:,n^{l+1})} = \frac{\hat{\mathbf{U}}^l_{(:,n^{l+1},:)} \times \hat{\mathbf{U}}^{\mathrm{T}l}_{(:,n^{l+1},:)}}{\sqrt{d^l}} \tag{16}$$

Yukarıdaki katmanın her n^1+1 kapsülü için simetrik bir matris A^1:,:,n^1+1 içerir. √d^1 terimi eğitimi stabilize eder ve birleştirme katsayıları ile log öncelikleri arasında bir dengenin korunmasına yardımcı olur. Her self-attention matrisi, n^1 kapsül tahminlerinin her bir kombinasyonu için puan anlaşmasını içerir ve bu nedenle, tüm birleştirme katsayılarını hesaplamak için kullanılabilirler. Özellikle, Denklem 17, Denklem de kullanılabilecek son katsayıları hesaplamak için kullanılır.

$$\mathbf{A}^l_{(:,:,n^{l+1})} = \frac{\hat{\mathbf{U}}^l_{(:,n^{l+1},:)} \times \hat{\mathbf{U}}^{\mathrm{T}l}_{(:,n^{l+1},:)}}{\sqrt{d^l}} \tag{17}$$

Böylece, l katmanındaki bir kapsül ile yukarıdaki katmandaki tüm kapsüller arasındaki bağlantı katsayıları, l + 1, toplamı bir olur. Ardışık olarak, son yönlendirme ağırlıklarını elde etmek için ilk log öncesi olasılıkları birleştirme katsayılarına eklenir. Daha derin bir hiyerarşi oluşturmak için üst üste yığılmış çoklu kapsül katmanlarının varlığında prosedür değişmeden kalır.

Sonuç

Doğru çalışan bir kapsül ağının, bilgileri daha iyi ve verimli bir şekilde yerleştirme becerisi sayesinde çok daha düşük sayıda parametreyle daha yüksek sonuçlar elde etmesini sağlamayı amaçlıyoruz. Bu bölümde, önerilen metodolojiyi deneysel bir bağlamda test ediyoruz ve genel becerilerinin ve verimliliğinin geleneksel convolutional (neural networks) sinir ağlarına ve literatürde bulunan benzer çalışmalara göre verimliliğini değerlendiriyoruz. Bu amaçla, önerilen metodolojimizi kapsül tabanlı ağ değerlendirmesi için Yale veri setiyle test ediyoruz.

1.5 Dense Capsule Networks (DcNet)

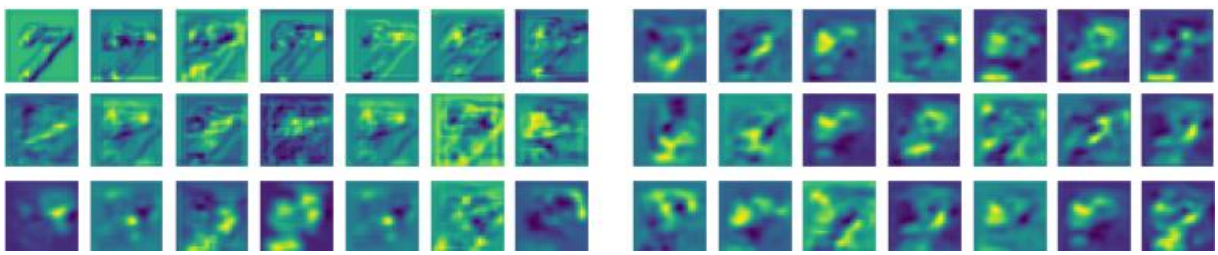
Son yıllarda, derin ağlar görüntü sınıflandırma, nesne tanıma gibi zorlu görevlere uygulanmış ve önemli gelişmeler göstermiştir. CNN'nin birçok çeşidi, zaman içinde daha derin ve daha karmaşık hale getirilerek önerilmiştir. Çeşitli kombinasyonlarda daha fazla derinlik eklemek, performansta önemli bir iyileşmeye yol açmıştır. Ancak derinlikteki artış, kaybolan gradyan problemine yol açar. Bu sorun ResNets , FractalNets tarafından ilk katmanlardan sonraki katmanlara bağlantılar eklenerek giderildi. Atlama bağlantılarının eklenme şeklini daha da basitleştiren bir başka benzer yapı, Huang ve diğerleri tarafından önerilmiştir. Topluluk

arasında DenseNets olarak bilinir. Ağ, ileri beslemeli bir şekilde diğer her katman arasında yoğun bağlantılar ekler. Bu yoğun bağlantıların eklenmesi, geleneksel CNN ile karşılaştırıldığında daha az sayıda parametreye yol açar. Bu özellik haritalarını birleştirmenin bir başka yararı, daha derin ağların eğitilmesine olanak tanıyan ağ boyunca daha iyi gradyan akışıdır.

CNN'ler, çeşitli bilgisayarlı görme ve makine öğrenimi görevlerinde gerçekten iyi performans gösterdi, ancak Sabour ve diğerleri tarafından vurgulanan birkaç dezavantajı var. Birincisi, CNN'lerin dönüşümlere karşı dayanıklı olmamasıdır, yani nesne konumunda hafif bir kayma, CNN'lerin tahminlerini değiştirmesine neden olur. Bu sorun, eğitim sırasında veri artırma yoluyla bir dereceye kadar azaltılabilsede, bu, ağı test verilerinde bulunabilecek herhangi bir yeni poz veya varyasyona karşı sağlam kılmaz. Diğer bir önemli dezavantaj, genel CNN'lerin herhangi bir karar verirken bir görüntüdeki nesneler arasındaki uzamsal ilişkileri dikkate almamasıdır. Basitçe açıklamak gerekirse, CNN'ler bir karar vermek için bir görüntüdeki yalnızca belirli yerel nesnelerin varlığını kullanırken, aslında mevcut nesnelerin uzamsal bağlamı eşit derecede önemlidir. Bunun nedeni, esas olarak ağda gerçekleştirilen, özelliklerin varlığına önem veren ve özelliklerin konum bilgilerini göz ardı eden, ağ büyüdükçe parametreleri azaltmak için yapılan havuzlama işlemidir.

Bu dezavantajların üstesinden gelmek için kapsül ağları (CapsNet) adı verilen bir seminal mimari önerdi. Bu modelde, bilgiler skaler (basit sinir ağlarında olduğu gibi) yerine vektör düzeyinde depolanır ve birlikte hareket eden bu nöron grubu kapsül olarak adlandırılır. MNIST veri kümesinde en son teknolojiye sahip doğruluğu elde etmek ve yeniden yapılanma düzenlemesini kullanarak çakışan rakamları daha iyi bir şekilde tespit etmek için anlaşma ve katman tabanlı ezme yoluyla yönlendirme kavramı kullanıldı. Capsnet'ler gerçekten güçlüdür, ancak aynı zamanda, geliştiriciler herhangi bir havuzlama katmanı ve derinliği kullanmadıkları için karmaşıklık açısından iyileştirme kapsamı vardır, çünkü ağ şu anda yalnızca bir kat konvolüsyon ve kapsül kullanmaktadır. Öte yandan, DenseNetler özellik birleştirme ile çok yüksek performans elde etme yeteneğine sahiptir. Özellik birleştirme özelliğine ve 8 kat konvolüsyona sahip Yoğun bir Konvolüsyon Ağını,birleştirme olmadan 8 kat konvolüsyona sahip basit bir Cnn'yi ve her katmanda 32 çekirdeği karşılaştırıldı. Hem nicel analiz (Tablo 2) hem de görselleştirmeler (Şekil 15), DenseNetlerin çeşitlendirilmiş özellikleri yakalamada daha iyi olduğunu göstermektedir. Bu özellik birleştirme fikrini DenseNet'lerden katmanlar arasında ödünç alıyoruz, aksi takdirde çok daha derin bir ağ gerektiren çeşitlendirilmiş özellikleri öğrenme potansiyeline sahip olduğu için bu kavramı dinamik yönlendirme algoritmasına girdi olarak kullanıldı.

Ayrıca, yeniden yapılandırma çıktılarında iyileştirme ile sonuçlanan birleştirilmiş yoğun katmanlara sahip değiştirilmiş bir kod çözücü ağı tasarlamak için DenseNets'in arkasındaki sezgi takip ediliyor. Sonuçlar(Tablo 1), geleneksel CapsNet'i eğitmek için 1000 epoch gerektiren 50 epoch MNIST veri setindeki son teknoloji performansla eşittir. Ayrıca, önerilen yöntemi FashionMNIST , The Street View House Numbers (SVHN) , AffNIST Veri Kümesi ve beyin tümörü veri kümesi gibi çeşitli sınıflandırma veri kümeleri üzerinde değerlendirildi ve performansı karşılaştırıldı.



(A) (B)

Şekil 15: Burada (A) ve (B), sırasıyla DenseNets ve CNN'den gelen aynı derinlikteki özellik haritalarıdır. DenseNets'in CNN'ye kıyasla daha net ayırt edici özellikler öğrendiği açıktır. (https://github.com/ssrp/Multi-level-DCNet)

50 epochs için Capsule Network ve DenseCapsNet öğrenme oranı, öğrenme bozulma oranı ve kapsül parametrelerinin sayısını aynı tutar.

Arka Plan

CNN için literatür çok geniştir. Konvolüsyonlar kullanılarak önerilen mimariler, hesaplama gücündeki artış nedeniyle önemli ölçüde artmıştır. CNN, alt katmanların kenarlar gibi temel özellikleri öğrendiği alt katmanlardan üst katmanlara doğru hiyerarşik bir şekilde öğrenmeye çalışır ve üst katmanlar bu düşük seviyeli özelliklerin birleşimiyle karmaşık özellikleri öğrenir. Daha derin ağlar performansta iyileşmeye yol açsada, parametre sayısındaki büyük artış nedeniyle eğitilmesi çok daha zordur. Önerilen son mimariler, parametre sayısını birlikte optimize ederken performansı iyileştirmeyi amaçlar. Otoyol ağı [7], çok sayıda katmanla daha derin ağı eğitmek için bu yönde önerilen ilk mimariydi. Modeli kolayca eğitmek için atlama yolları eklediler. ResNet [4] modeli, artık bağlantılar ekleyerek eğitimi iyileştirir. Huang ve arkadaşları tarafından önerilen bu tür başka bir ağ [2], ilk Konvolüsyon katmanlarından daha derin katmanlara bağlantılar ekleyerek, onu tek bir yoğun blok olarak adlandırarak atlama bağlantıları eklemenin yeni bir yolunu yarattı.

Kapsül ağları , CNN'lerin dezavantajlarının üstesinden gelmek için yakın zamanda tanıtılmıştır. Kapsüller, bir görüntüde bulunan çeşitli varlıkların özelliklerini gösteren nöron grubudur. Bir görüntünün konum, boyut, doku gibi yakalanabilecek çeşitli özellikleri olabilir. Kapsüller, çıktının tüm nihai kapsüllere gönderildiği anlaşmaya göre yönlendirmeyi kullanır. Her kapsül, daha sonra ana kapsülün gerçek çıktısıyla karşılaştırılan ana kapsül için bir tahmin yapar. Çıktılar eşleşirse, iki kapsül arasındaki bağlantı katsayısı artar. u_i bir i kapsülünün çıktısı ve j ana kapsül olsun, tahmin şu şekilde hesaplanır:

$$\hat{u}_{i|j} = \mathbf{W}_{ij} u_i \qquad c_{ij} = \frac{exp(b_{ij})}{\sum_k exp(b_{ik})} \tag{18}$$

burada W_{ij}ağırlıklandırma matrisidir. Daha sonra c_{ij} kuplaj katsayıları gösterildiği gibi basit bir softmax fonksiyonu kullanılarak hesaplanır. Burada b_{ij}, i kapsülünün j kapsülü ile birleştirilmesinin log olasılığıdır. Yönlendirme başlatıldığında bu değer 0'dır. Ana kapsül j'ye giriş vektörü şu şekilde hesaplanır:

$$s_j = \sum_i c_{ij} \hat{u}_{j|i} \qquad v_j = \frac{||s_j||^2}{1 + ||s_j||^2} \frac{s_j}{||s_j||} \tag{19}$$

Bu kapsül vektörlerinin çıktısı, kapsül tarafından temsil edilen bir nesnenin verilen girdide bulunup bulunmadığı olasılığını temsil eder. Ancak bu kapsül vektörlerinin çıktısı, çıktıya bağlı olarak birini aşabilir. Bu nedenle, vektör uzunluğunu 1 ile sınırlamak için yukarıda tanımlanan doğrusal olmayan bir ezme işlevi kullanılır, burada s_j j kapsülüne girdidir ve v_j çıktıdır. Günlük olasılıkları, v_j ve $u_{i|j}$'nin iç çarpımı hesaplanarak güncellenir. İki vektör aynı fikirdeyse, ürün daha büyük olur ve bu da daha uzun vektör uzunluğuna yol açar. Son katmanda, her kapsül için bir kayıp değeri hesaplanır. Varlık olmadığında kayıp değeri yüksektir ve kapsül yüksek örnekleme parametrelerine sahiptir. Kayıp şu şekilde tanımlanır:

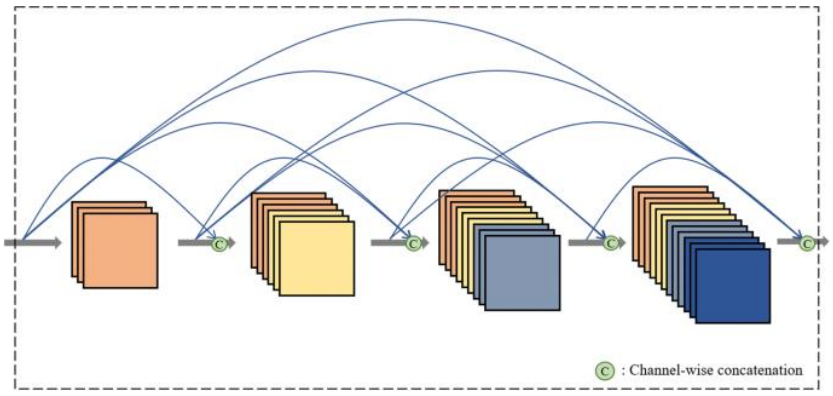
$$L_k = T_k \, max(0, \, m^+ - ||v^k||)^2 + \lambda (1 - T_k) \, max \, (0, ||v_k|| - m^-)^2$$

Burada L_k, bir k kapsülü için kayıp fonksiyonudur T_k, etiket doğru olduğunda 1, aksi halde 0'dır. CapsNet bir konvolüsyon uygular ve 256 özellik haritası oluşturur. Daha sonra, adım 2 ile 9 x 9 çekirdek kullanılarak 32, 8D kapsüllerin oluşturulduğu birincil kapsül katmanına beslenir, ardından ezilir.
Son olarak., 16D'nin son kapsüllerini oluşturan bir DigitCaps katmanı uygulanır. Afshar ve arkadaşları [6], kapsül ağlarının karmaşık veri kümeleri içeren görüntüler üzerinde çalışacak kadar güçlü olduğunu göstermektedir. DenseNet kullanarak kapsülleri güçlendirme hedeflenir.

Dense Capsule Networks

Temel CapsNet modelindeki ilk konvolüsyon katmanı tarafından öğrenilen özellik haritaları, karmaşık veri kümeleri için kapsüller oluşturmak için yeterli olmayabilecek çok temel özellikleri öğrenir. Bu nedenle, ilk katmanda konvolüsyon katmanlarını ikiye ve sekize çıkarmayı denendi ve Dcnet'te Tablo 1'de gösterildiği gibi herhangi bir iyileşmeye yol açmadığını gözlemlendi, kapsül ağlarını daha derin bir mimari oluşturacak şekilde değiştirmeye çalışıldı. Atlama bağlantılarına dayalı sekiz katmanlı yoğun konvolüsyonel alt ağ oluşturuyoruz. Her katman, ileri beslemeli bir şekilde bir sonraki katmana birleştirilir ve son bir evrişim katmanı oluşturmak için eklenir. Bu, doğrudan yığılmış konvolüsyon katmanlarına kıyasla daha iyi gradyan akışına yol açar.

Şekil 16 Dense Bloğundaki yoğun bağlantıyı göstermektedir:

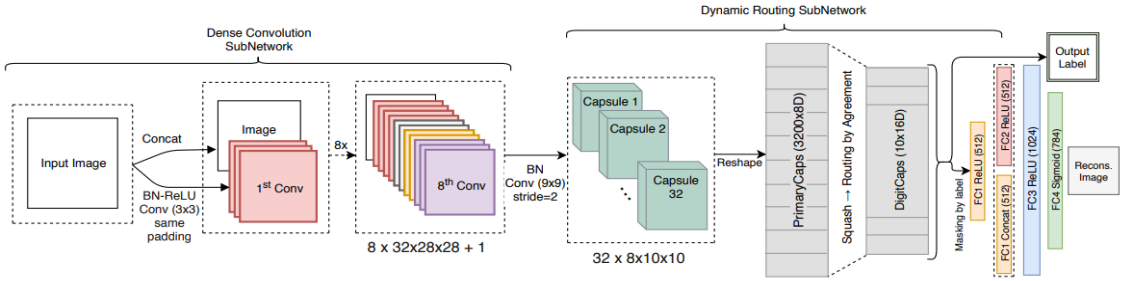


Şekil 16: DenseNet'te yoğun bağlantı mekanizması
(Guangcong Sun,Shifei Ding,Tongfeng Sun,Chenglong Zhang,Wei Du,2021)

Dense Capsule aracılığıyla çok seviyeli özellik yeniden kullanımını sağlayan özellik-kapsül(feature-capsule) gösterimini kullanarak kapsül ağındaki görüntü sınıflandırma görevlerinin işlenmesi optimize edilir. Özellik-kapsül birleştirme sürecinde, özellik haritası ve özellik kapsülü arasındaki farkı göz önünde bulundurarak, sonraki katmanlarda girdi şansını artırmak için bitişik katmanlar tarafından öğrenilen özellik kapsülleri birleştirilir. Bu yönteme kapsül şeklinde birleştirme denilir. Kapsül tabakasının çıktısı için $l \;\; \emptyset^l \;\; \in R^{(w^l, w, c^l, n^l)}$ atlama bağlantısından sonra l + 1 katmanının çıktısı $\emptyset^{l+1} \;\; \in R^{(w^{l+1}, w^{l+1}, c^{l+1}, 2n^{l+1})}$ burada w^{l+1} , c^{l+1},

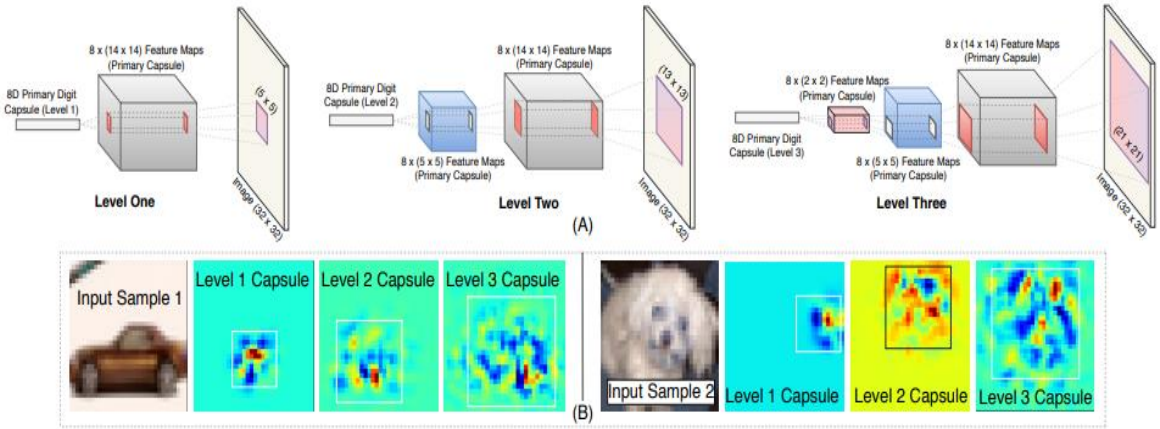
n^{l+1} ve w^l, c^l , n^leşittir. Her blokta, $\frac{L(L+1)}{2}$ kapsül katmanları arasında bağlantılar oluşturulur .

DenseCaps, girdi olarak görüntüleri ve veri akışının birimi olarak kapsülleri alır. Özelliğin yeniden kullanımına dayalı olarak eğitilerek çok seviyeli kapsül füzyonunun özellik görüntü sınıflandırma görevleri için daha uygun hale getirir



Şekil 17: MNIST için önerilen tahmin pipeline düzeni (https://github.com/ssrp/Multi-level-DCNet)

Şekil 17, MNIST veri kümesi için önerilen mimarinin ayrıntılı pipeline hattını göstermektedir. Giriş örneği 8 kıvrım seviyesine gider ve bu kıvrım seviyelerinin her biri 32 yeni özellik haritası oluşturur ve ardından önceki tüm katmanların özellik haritalarıyla birleştirilerek 257 özellik haritası elde edilir (giriş görüntüsü dahil edilmiştir). Bu çeşitlendirilmiş özellik haritaları, 2'lik bir adımla 9 × 9'luk bir konvolüsyon uygulayan kapsül katmanına girdi görevi görür. Elde edilen özellik haritaları, Capsnet'in birincil kapsülleri olarak işlev görür. Sabour[9] esas olarak tamamen aynı fikirde olunan değişmezlik yerine eşitliğe odaklanır, bu yüzden maks. veya DenseNetler'de kullanılan ortalama havuz katmanları, uzamsal bilgi kaybına neden olur. Sabour ve arkadaşları birincil kapsülleri, aynı karmaşıklık düzeyinde evrişim ile oluşturulan 256 özellik haritasından oluşturan DcNet'in birincil kapsülleri, farklı karmaşıklık seviyelerinin tüm özelliklerini birleştirerek üretilir ve bu da sınıflandırmayı daha da geliştirir.
Bu özellik haritaları, bir squash aktivasyon katmanına geçirilen otuz iki 8D kapsülü ve ardından yönlendirme algoritması olarak işlev görür. MNIST için kullanılan geleneksel kapsül ağına benzer şekilde, one-hot 10D çıkış vektörü üreten 10 sınıfın (basamak) her biri için 16D nihai kapsüller üretilir.



Şekil 18: Farklı seviyelerde karşılık gelen kapsüller tarafından CIFAR-10'da etkinleştirilen görüntü bölgesi. Satır A, giriş görüntüsünde birincil kapsülün nasıl etkinleştirildiğini gösterir. Satır B, DCNet'de farklı düzeylerdeki kapsüllerden alınan giriş görüntüsünde örnek

etkinleştirme bölgelerini gösterir(Sai Samarth R. Phaye, Apoorva Sikka, Abhinav Dhall, Deepti Bathula,2018)

Huang ve arkadaşları tarafından uygulanan yoğun bağlantılardan esinlenerek, kapsül ağının yeniden yapılandırma modeli de değiştirildi. Kod çözücü(Decoder), ilk katman ve ikinci katmanın özelliklerinin birleştirilmesiyle daha iyi yeniden yapılandırmalar sağlayan, DigitCaps katmanını girdi olarak alan (eğitim sırasında çıktı etiketiyle maskelenen) dört katmanlı bir modeldir. Görüntünün boyutu 32×32'den büyükse, nöron sayısı 512'den 600'e ve 1024'ten 1200'e değiştirilir. Deneyler, MNIST, Fashion-MNIST, SVHN gibi çeşitli veri kümeleri üzerindeki performanslarda önemli artışlar göstermektedir. DCNet'in CIFAR-10 veri kümesindeki performansının, aynı parametrelerle eğitilmiş tek temel CapsNet modeline göre arttığı fark edildi, ancak %89,40 doğruluğa sahip kapsül ağların yedi topluluk modelinden [9] daha iyi performans göstermedi. CIFAR-10'daki görüntüler MNIST veri seti ile karşılaştırıldığında oldukça karmaşık olduğundan, ağın parça-bütün ilişkilerini kodlaması kolay değildir. Bu sorunu, bu tür karmaşık veri kümelerini iyi öğrenen DCNet++ oluşturarak çözebiliriz.

Deneyler

Modelimizin potansiyelini birden çok veri kümesinde gösteriyoruz ve onu CapsNet mimarisiyle karşılaştırıyoruz.Tüm modeller 50 - 100 epoch için çalıştırıldı. Optimize edici olarak ilk öğrenme oranı 0,001'e ve bozulma oranı 0,9'a ayarlandı. Çarpan faktörünü, marj kaybına baskın gelmemesi için görüntü boyutuna göre yeniden yapılandırma kaybını küçültmek için değiştirildi.Test hataları her model için bir kez hesaplanmıştır. Tüm deneyler için üç rota kullanıldı. Adil karşılaştırmalar için, geleneksel CapsNet ile aynı olan PrimaryCaps katmanından sonra önerilen DCNet modelinin parametrelerini çoğunlukla tuttuk. Kullanılan veri kümelerine karşılık gelen uygulama ayrıntıları aşağıdadır:

MNIST El yazısı rakamlar veri seti ve Fashion-MNIST veri seti, her biri 28×28 boyutunda olan 60K ve 10K eğitim ve test görüntülerine sahiptir. Herhangi bir veri artırma şeması kullanılmadı ve deney 3 kez tekrarlandı. Tablo 1'de, DCNet'in CapsNet'e kıyasla giriş verisi varyasyonlarını hızlı bir şekilde öğrenebildiği, yani MNIST üzerinde %99,75 ve FashionMNIST veri setinde %94,64 oranında 20 kat azalma ile test doğruluğu elde edebildiği Tablo 1'de açıkça görülmektedir.

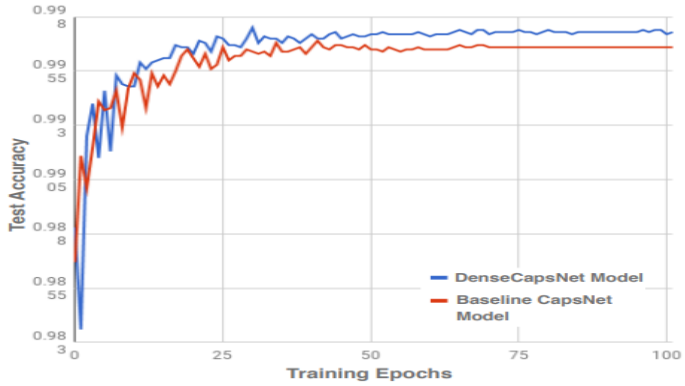
Model	MNIST	Fashion-MNIST	SVHN	SmallNORB	Brain Tumor Dataset	CIFAR-10
Baseline CapsNet	99.67% (50E) 99.75% (1000E) [■] 99.65% (1000E, NR) [■]	93.65% (100E)	93.23% (100E) 95.7% (> 100E) [■]	89.56% (50E)	78% (10E)[■] 87.5% (50E)	89.40% [■] (7 model ensemble)
DCNet	**99.75% (50E)** **99.71% (50E, NR)** **99.72% (50E, BR)**	94.64% (100E) 94.59% (100E, NR)	95.58% (50E)	94.43% (50E)	86.8% (10E) 93.04% (50E)	82.63% (single model)

Tablo 1: Çeşitli veri kümeleri üzerinden CapsNets, performansı. (Sai Samarth R. Phaye, Apoorva Sikka, Abhinav Dhall, Deepti Bathula,2018)

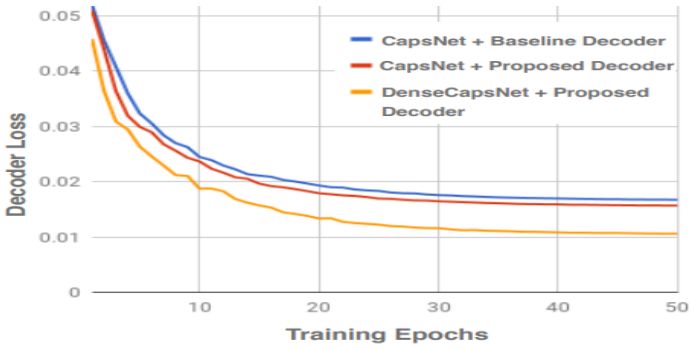
Model	Description	Parameters	Test Acc.
CapsNet (Baseline)	Conv - Primary Capsules - Final Capsules	8.2M	99.67%
CapsNet Variant	Added one more initial convolution layer having 256 9x9 kernels with same padding, stride=1	13.5M	99.66%
DCNet Variant One	Used 3 convolution layers with 8 feature maps each	6.9M	99.66%
DCNet Variant Two	Removed the concatenations (no skip connections) acting like simple 8 layered convolutions	4M	99.68%
DCNet Variant Three	Used 8th convolution layer only (no concatenation in the last layer) to create Primary Capsules	7.2M	**99.72%**
Final DCNet	DenseConv - Primary Capsules - Final Capsules	11.8M	**99.75%**

Tablo 2: MNIST veri setinde 50 epochtan sonra çeşitli model varyasyonlarının karşılaştırılması. (Sai Samarth R. Phaye, Apoorva Sikka, Abhinav Dhall, Deepti Bathula,2018)

Street View House Numbers (SVHN), eğitim ve test için 73K ve 26K gerçek hayattaki rakam görüntüleri içerir. İlk 50 epoch için sonuçları orijinal kapsül ağ modeli [12] ile karşılaştırmak için, 18 özellik haritasına sahip dört evrişim katmanından 8D nihai kapsül ile 6D 16 birincil kapsül kullanarak DCNet'i değiştirdik. DCNet modelinin sonuçları, kopyalanan CapsNet modeline göre iyileşerek %93,23'ten %95,59'luk doğruluğu test etti.
Beyin Tümörü Veri Kümesi, üç beyin tümörü türünden biri (yani Meningioma, Glioma ve Hipofiz tümörü) teşhisi konan 233 hastanın 3.064 MRI görüntüsünü içerir. Afshar ve arkadaşları [5], 64 özellik haritasına sahip ilk evrişim katmanıyla kapsül ağ mimarisini değiştirdi. Sekiz ilk evrişim katmanını her biri 16 çekirdekli dört katmana değiştirerek eşdeğer bir DCNet modeli oluşturuldu (tek tip sonuçlar için), toplam 64 çekirdeğe ve birincil kapsülleride 6'ya düşürüldü Ayrıca modelin öğrenme oranı 1E olarak değiştirildi. Modelleri sekiz aşamalı çapraz doğrulama konusunda eğitildi (sonuçlar Tablo 1'de).



Şekil-19: DCNet ve CapsNet modelinin MNIST veri kümesindeki performansının karşılaştırılması (Sai Samarth R. Phaye, Apoorva Sikka, Abhinav Dhall, Deepti Bathula,2018)



Şekil-20: MNIST veri kümesinde CapsNet+Baseline Kod Çözücü, CapsNet+Önerilen Kod Çözücü ve DCNet+Önerilen Kod Çözücünün Yeniden Oluşturulması MSE(OHK) kaybı. (Sai Samarth R. Phaye, Apoorva Sikka, Abhinav Dhall, Deepti Bathula,2018)

SmallNORB veri kümesi, eğitim ve test için 24K ve 24K görüntülere sahiptir. Görüntüler normalleştirildi ve her görüntüye rastgele gürültü ve kontrast eklendi. Önerilen model, her biri 18 çekirdekli dört evrişim katmanından 8D son kapsül ile 6D 16 birincil kapsül alarak DCNet modifiye edilerek ilk 50 epoch için CapsNet ile karşılaştırıldı. DCNet'in sonuçları, %95,58'lik bir test doğruluğu sağlamak için çoğaltılan CapsNet modeline göre iyileştirildi.
DCNet'teki değiştirilmiş kod çözücünün kaybının, CapsNet ile aynı kayıp çarpan faktörüne sahip olarak önemli miktarda azaldığını bulduk. Ayrıca Şekil 19, DCNet'in daha hızlı bir yakınsama oranına sahip olduğunu açıkça çıkarabileceğimiz CapsNet ve DCNet'in test doğruluğu karşılaştırmasını göstermektedir.

Sonuç

CapsNet'teki standart evrişim katmanlarını yoğun bağlantılı evrişim ile değiştiren bir Capsule Network, DCNet modifikasyonu önerdik. Ardışık iki katman arasındaki doğrudan bağlantıların eklenmesi, daha iyi özellik haritalarının öğrenilmesine yardımcı olur ve bu da daha kaliteli birincil kapsüllerin oluşturulmasına yardımcı olur. Bu mimarinin etkinliği, geleneksel CapsNet'lere göre toplam eğitim yinelemelerinde yirmi kat azalma ile MNIST verilerindeki son teknoloji performans (%99,75) ile kanıtlanmıştır. Aynı DCNet modeli, CIFAR-10 verilerinde tek bir temel CapsNet modelinden daha iyi (%82,63) performans gösterse de, %89,40 doğrulukla CapsNet'in yedi topluluk modeline kıyasla eşitin altındaydı. CapsNet'in gerçek hayattaki karmaşık veriler (CIFAR-10, ImageNet, vb.) üzerindeki performansının, MNIST gibi daha basit veri kümelerine kıyasla standartların altında olduğu bilinmektedir.Önerilen ağlar, SVHN, SmallNORB ve Tümör Veri Kümesi gibi diğer veri kümeleri üzerindeki mevcut Kapsül Ağlarının performansını önemli ölçüde iyileştirir.

1.6 Veri Seti

Veri seti, belirli bir konunun sayılar veya değerler koleksiyonu olarak toplanıp saklanmasıyla oluşturulan dosyalara denir. Yapay Zekâ, insanların beyin çalışma mantığını makineye öğreterek, insanların düşünce ve temel becerilerini kazandırmak için yazılmış

algoritmalardır. Bu algoritmalar insanların öğrenme tekniği gibi dış dünyadaki olayları, cisimleri görme ve tanıma ihtiyacı duyarlar. Fakat insanlar bu olayı uzun süre boyunca yapar ve zihinlerinde biriken bilgileri, nesneleri ancak bu süreç sonunda kullanılabilir hale dönüştürebilir fakat bu durum yapay zekada söz konusu değildir. Yapay zekaya önceden toplanmış verileri anlık sisteme vermekle sonuç elde etmek mümkündür. Bu toplanmış verilere "Veri Seti" deriz. Yale Yüz Veri tabanı, 165 gri tonlamalı 15 kişilik GIF formatında görüntü içerir. Görüntüler gözlüklü, gözlüksüz, üzgün, uykulu, şaşırmış vb. farklı yüz ifadesi içermektedir. Veri kümesinin boyutu 6.4 MB'dir ve her biri 576 görüntüleme koşulunda görülen 10 nesnenin 5760 tek ışık kaynağı görüntüsünü içerir.

Şekil 21: Veri setinden örnek görüntüler.
(https://www.kaggle.com/datasets/olgabelitskaya/yale-face-database).

2.YAPILAN ÇALIŞMALAR

2.1 OrjinalCapsnetin Yale VeriSeti Üzerinde Uygulanması

Bu çalışmamızda yale veri setini kullanarak OrjinalCapsnet yöntemini pytorch framework'ü ile gerçeklenmesini ele alacağız.

Gerekli kütüphane fonksiyonlarının import edilmesi

```
import torch.utils.data as torchdata
from pathlib import Path
import cv2
import glob
from timm.utils import AverageMeter, CheckpointSaver, NativeScaler
from sklearn.metrics import accuracy_score
from PIL import Image
import warnings
from torchvision import transforms, utils
from torch.utils.data import Dataset, DataLoader
import matplotlib.pyplot as plt
from skimage import io, transform
import pandas as pd
import os
```

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
from torch.optim import Adam
from torchvision import datasets, transforms
USE_CUDA = False

warnings.filterwarnings("ignore")
plt.ion()
```

İkinci aşama olan veri setimizin ölçeklendirme ve normalize işlemlerini gerçekleştirelim. Veri ölçeklendirmemizin amacı girdilerimizin birbirleriyle aynı aralıktaki sayılardan oluşturmaktır.
Class Yale ismindeki sınıfımız ile yale veri seti üzerinde gerekli işlemleri yaparak hazır hale getiriyoruz.

```
class TrainDataset(torchdata.Dataset):
    def __init__(self, fileList, labels):
        self.fileList = fileList
        self.labels = labels

    def __len__(self):
        return len(self.fileList)

    def __getitem__(self, index: int):
        filename = self.fileList[index]
        img = Image.open(filename)
        img = np.array(img.resize((28, 28), Image.ANTIALIAS))
        # img = img / 255  ## normalize
        dataset_transform = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize((0.1307,), (0.3081,))
        ])
        img = dataset_transform(img)
        label = torch.sparse.torch.eye(15).index_select(
            dim=0, index=torch.tensor(self.labels[index]))

   return img, label


class Yale:
    def __init__(self, _batch_size, batchSizeTest, trainIndex, testIndex):
        train_dataset =
TrainDataset(np.array(fileLinks)[trainIndex.astype(int)], labels[trainIndex])
        test_dataset = TrainDataset(np.array(fileLinks)[testIndex.astype(int)],
labels[testIndex])
        self.train_loader = torch.utils.data.DataLoader(train_dataset,
batch_size = _batch_size, shuffle=True)
        self.test_loader = torch.utils.data.DataLoader(test_dataset, batch_size
= batchSizeTest, shuffle=False)
```

Giriş katmanı modelin ilk katmanıdır. Giriş olarak alınan bu veriler evrişim katmanına

gönderilir. Öncelikle girişlerimiz bu Convlayer katmanına gönderilir ve bu katman evrişimli sinir ağında bulunan evrişim katmanlarının aynısıdır.

```
class ConvLayer(nn.Module):
    def __init__(self, in_channels=1, out_channels=256, kernel_size=9):
        super(ConvLayer, self).__init__()

        self.conv = nn.Conv2d(in_channels=in_channels,
                              out_channels=out_channels,
                              kernel_size=kernel_size,
                              stride=1
                              )

    def forward(self, x):
        return F.relu(self.conv(x))
```

Aşağıdaki çıktı görüntüsünde görüldüğü üzere x bizim ConvLayer katmanımıza giren girdi değerlerimizdir. Batchsize 22, görüntülerimiz gri tonlamada olduğundan dolayı 1dir. 28,28 ise ilk etapta image size'a verdiğimiz değerlerdir.
Çıkış değerimizde ise bu değer 256 olmuştur.

```
IPdb [2]: x.shape
torch.Size([22, 1, 28, 28])

IPdb [3]: F.relu(self.conv(x)).shape
torch.Size([22, 256, 20, 20])
```

Görüntü verilerinin boyutu evrişim katmanından sonra (çekirdek boyutu- 1 = 8) 8 piksel kaybeder. Uygulanan evrişim katmanından sonra görüntü boyutları 20x20 piksele düşmüştür. Daha sonra elde edilen bu görüntülere özellik kapsülü(primary) katmanı uygulanmıştır.

Bu katman evrişim katmanı çıkışlarını giriş olarak almaktadır. Evrişim katmanından farklı olarak bu katmanda boyutlandırma işlemi ve squash fonksiyonu uygulanmaktadır. Evrişim katmanıyla 20x20 boyutlarına indirgenen görüntünün primary kapsülü katmanının kaydırma adımı 2 olan parametresiyle boyutları 10x10 boyutlarına düşürülmüştür. Bununla beraber çekirdek boyutu 9 alınarak (çekirdek boyutu-1=8/2=4) görüntü 4 piksel daha kaybedip bu katman çıkışında 6x6 boyutuna gelmiştir. Daha sonra bu görüntüleri 32 boyutlu vektörler haline getirmek için yeniden bir boyutlandırma işlemi uygulayarak görüntüler vektör boyutuna indirgenmiştir.

```
class PrimaryCaps(nn.Module):
    def __init__(self, num_capsules=8, in_channels=256, out_channels=32,
kernel_size=9):
        super(PrimaryCaps, self).__init__()

        self.capsules = nn.ModuleList([
            nn.Conv2d(in_channels=in_channels, out_channels=out_channels,
                      kernel_size=kernel_size, stride=2, padding=0)
            for _ in range(num_capsules)])
```

```
    def forward(self, x):
        u = [capsule(x) for capsule in self.capsules]
        u = torch.stack(u, dim=1)
        u = u.view(x.size(0), 32 * 6 * 6, -1)
        return self.squash(u)

    def squash(self, input_tensor):
        squared_norm = (input_tensor ** 2).sum(-1, keepdim=True)
        output_tensor = squared_norm * input_tensor / \
            ((1. + squared_norm) * torch.sqrt(squared_norm))
        return output_tensor
```

```
IPdb [6]: x.shape
torch.Size([22, 256, 20, 20])

IPdb [7]: u[0].shape
torch.Size([22, 32, 6, 6])
```

Bu katmandaki kapsül sayısı tahmin edilmesi gereken sınıfların sayısı ile belirlenir. Ayrıca bu katmanda bulunan kapsüller, özellik yönelimlerine dayalı olarak sınıflar hakkında tahminlerde bulunacaktır fakat bu tahminlerin öncül kapsüller tarafından yapılan tahminlere uyması gerekir ve bu yöntem, dinamik yönlendirme algoritması ile sağlanır.

```
class DigitCaps(nn.Module):
    def __init__(self, num_capsules=15, num_routes=32 * 6 * 6, in_channels=8,
out_channels=16):
        super(DigitCaps, self).__init__()

        self.in_channels = in_channels
        self.num_routes = num_routes
        self.num_capsules = num_capsules

        self.W = nn.Parameter(torch.randn(
            1, num_routes, num_capsules, out_channels, in_channels))

    def forward(self, x):
        batch_size = x.size(0)
        x = torch.stack([x] * self.num_capsules, dim=2).unsqueeze(4)

        W = torch.cat([self.W] * batch_size, dim=0)
        u_hat = torch.matmul(W, x)

        b_ij = Variable(torch.zeros(1, self.num_routes, self.num_capsules, 1))
        if USE_CUDA:
            b_ij = b_ij.cuda()

        num_iterations = 3
        for iteration in range(num_iterations):
            c_ij = F.softmax(b_ij)
            c_ij = torch.cat([c_ij] * batch_size, dim=0).unsqueeze(4)
```

```
            s_j = (c_ij * u_hat).sum(dim=1, keepdim=True)
            v_j = self.squash(s_j)

            if iteration < num_iterations - 1:
                a_ij = torch.matmul(u_hat.transpose(
                    3, 4), torch.cat([v_j] * self.num_routes, dim=1))
                b_ij = b_ij + a_ij.squeeze(4).mean(dim=0, keepdim=True)

        return v_j.squeeze(1)

    def squash(self, input_tensor):
        squared_norm = (input_tensor ** 2).sum(-1, keepdim=True)
        output_tensor = squared_norm * input_tensor / \
            ((1. + squared_norm) * torch.sqrt(squared_norm))
        return output_tensor
```

```
IPdb [14]: x.shape
torch.Size([22, 1152, 8])
```

Decoder, 32 boyutlu bir vektörü doğru Digit kapsülden alır ve onu etiketli bir görüntüye dönüştürmeyi öğrenir. Decoder bir düzenleyici olarak kullanılır, doğru Digit kapsülün çıktısını girdi olarak alır ve 28'e 28 piksellik bir görüntüyü yeniden oluşturmayı amaçlar.
Kayıp işlevi, yeniden yapılandırılmış görüntü ile giriş görüntüsünün arasındaki Öklid mesafesidir. Decoder, kapsülleri orijinal görüntüyü yeniden oluşturmak adına yararlı olan özellikleri öğrenmeye zorlar. Yeniden oluşturulmuş görüntü, giriş görüntüsüne ne kadar yakınsa o kadar başarılıdır.

```
class Decoder(nn.Module):
    def __init__(self):
        super(Decoder, self).__init__()

        self.reconstraction_layers = nn.Sequential(
            nn.Linear(16 * 15, 512),
            nn.ReLU(inplace=True),
            nn.Linear(512, 1024),
            nn.ReLU(inplace=True),
            nn.Linear(1024, 784),
            nn.Sigmoid()
        )

    def forward(self, x, data):
        classes = torch.sqrt((x ** 2).sum(2))
        classes = F.softmax(classes)

        _, max_length_indices = classes.max(dim=1)
        masked = Variable(torch.sparse.torch.eye(15))
        if USE_CUDA:
            masked = masked.cuda()
        masked = masked.index_select(
            dim=0, index=max_length_indices.squeeze(1).data)

        reconstructions = self.reconstraction_layers(
            (x * masked[:, :, None, None]).view(x.size(0), -1))
        reconstructions = reconstructions.view(-1, 1, 28, 28)
```

```
        return reconstructions, masked
```

Oluşturduğumuz katmanları kullanarak modelimizin oluşturulması. Eğitim döngüsündeki her adımda, modelimizi veri yükleyiciden aldığımız örnekler üzerinde değerlendiririz. Daha sonra kayıp fonksiyonunu kullanarak modelimizin çıktılarını hedeflerle karşılaştırırız. Kayıp fonksiyonları ile gerçek çıktılarımızı ideal ile karşılaştırdıktan sonra, çıktılarını hedefe daha iyi benzetmek için modeli biraz zorlamamız gerekiyor.
Eğitim kaybı, modelimizin eğitim setine tam olarak uyup uymadığını bize söyleyecektir. Eğitim kaybı azalmıyorsa, model veri için çok basit olabilir. Diğer olasılık, verilerimizin anlamlı bilgiler içermemesidir

```
class CapsNet(nn.Module):
    def __init__(self):
        super(CapsNet, self).__init__()
        self.conv_layer = ConvLayer()
        self.primary_capsules = PrimaryCaps()
        self.digit_capsules = DigitCaps()
        self.decoder = Decoder()

        self.mse_loss = nn.MSELoss()

    def forward(self, data):
        output = self.digit_capsules(
            self.primary_capsules(self.conv_layer(data)))
        reconstructions, masked = self.decoder(output, data)
        return output, reconstructions, masked

    def loss(self, data, x, target, reconstructions):
        return self.margin_loss(x, target) + self.reconstruction_loss(data,
reconstructions)

    def margin_loss(self, x, labels, size_average=True):
        batch_size = x.size(0)

        v_c = torch.sqrt((x**2).sum(dim=2, keepdim=True))

        left = F.relu(0.9 - v_c).view(batch_size, -1)
        right = F.relu(v_c - 0.1).view(batch_size, -1)

        loss = labels * left + 0.5 * (1.0 - labels) * right
        loss = loss.sum(dim=1).mean()

        return loss

    def reconstruction_loss(self, data, reconstructions):
        loss = self.mse_loss(reconstructions.view(
            reconstructions.size(0), -1), data.view(reconstructions.size(0), -
1))
        return loss * 0.0005
```

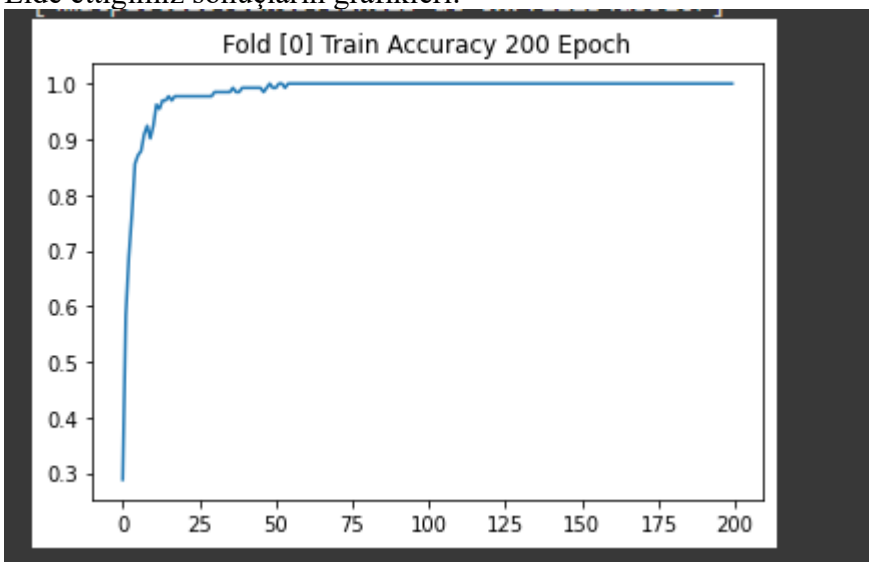
Yapılandırılan data setini data adı altındaki klasörden yükleme işlemini gerçekleştirip, data seti üzerinden cross validation'ı da uyguladık.
Cross-Validation, modelin görmediği veriler üzerindeki performansını olduğunca objectif ve doğru biçimde değerlendirmek için kullanılan yeniden örneklendirme yöntemidir diyebiliriz.
Eğer bir modelimizin başarısını tek bir train-test yaklaşımıyla ölçmek yerine k katlı cross-validation uygulamış olursak farklı bölünmeler için farklı accurracy skorları elde etmiş oluruz. Bu da modelimizi optimize etmeye ve model performansını arttırmamızı sağlar.
Peki bu Cross Validation nasıl çalışır?
1. Veri setimiz rastgele olacak şekilde karıştırılıyor.
2. Veri setimizi k tane gruba ayırırız.
3. Her grup için aşağıdaki adımlar gerçekleştirilir.
   - Seçilen grup validasyon seti olarak kullanılır.
   - Diğer tüm gruplar (k-1) train seti olur.
   - Train seti kullanılarak model kurulur ve validasyon seti ile değerlendirilir.
   - Modelin elde ettiği sonuç bir listede saklanır.

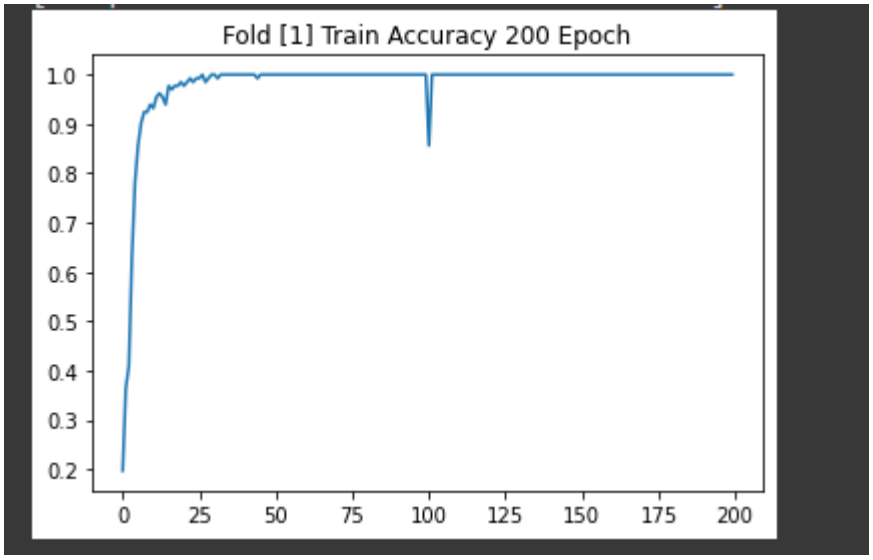
4. Daha sonra bu elde edilen değerlendirme puanlarının istatiksel özetine bakılır. Bu ortalama, standart sapma vs. gibi yöntemlerle sağlanabilir.

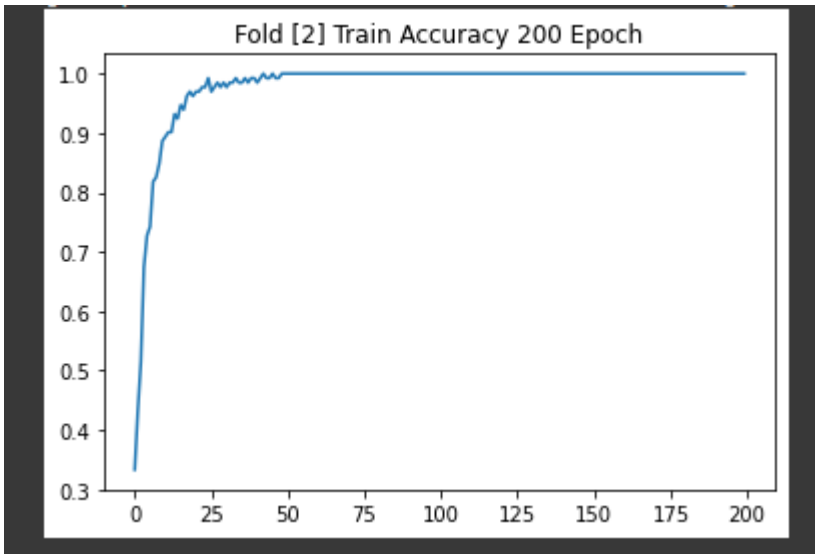
```
def build_dataset():
    fileList = []
    labels = []
    for i in range(1, 16):
        files = glob.glob('./data/subject'+str(i).zfill(2)+"*")
        for fname in files:
            fileList.append(fname)
            labels.append(i - 1)
    return fileList, np.array(labels)

batch_size = 22
batchSizeTest = int(batch_size / 2)
n_epochs = 200
finalAcc = np.zeros(5)
fileLinks, labels = build_dataset()
from sklearn.model_selection import StratifiedKFold
skf = StratifiedKFold(n_splits = 5)
for cv, (train_index, test_index) in enumerate(skf.split(fileLinks, labels)):

    yale = Yale(batch_size, batchSizeTest, train_index, test_index)
    capsule_net = CapsNet()
    if USE_CUDA:
        capsule_net = capsule_net.cuda()
    optimizer = Adam(capsule_net.parameters(), lr=0.0001)

    amp_autocast = torch.cuda.amp.autocast
    loss_scaler = NativeScaler()
    second_order = hasattr(optimizer, 'is_second_order') and
optimizer.is_second_order
    for epoch in range(n_epochs):

        print("Fold = ", str(cv), "Epoch = ", str(epoch + 1))
        capsule_net.train()
        train_loss = 0
        targetTensor = torch.tensor(np.zeros((132, 15)))
```

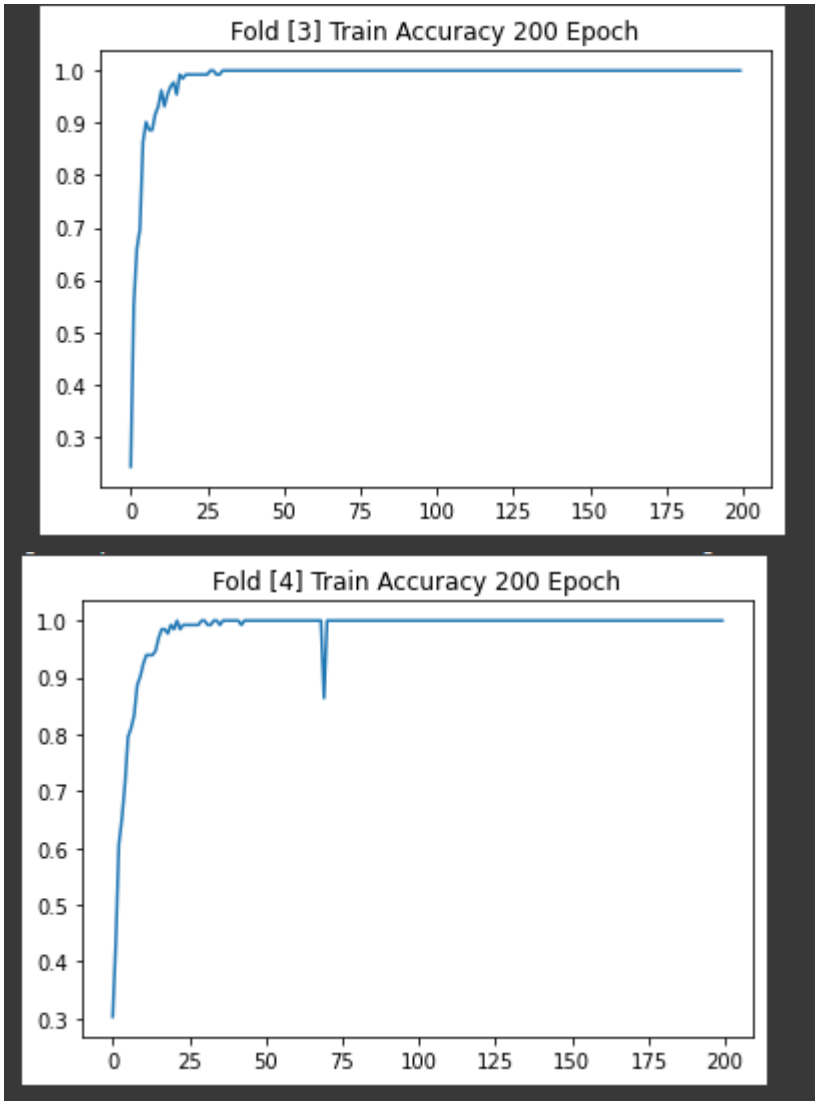
```
        maskedTensor = torch.tensor(np.zeros((132, 15)))
        for batch_id, (data, target) in enumerate(yale.train_load1er):

            data, target = Variable(data), Variable(target)
            target = target.view(target.shape[0], target.shape[2])
            if USE_CUDA:
                data, target = data.cuda(), target.cuda()
            with amp_autocast():
                output, reconstructions, masked = capsule_net(data.float())
                loss = capsule_net.loss(data, output, target.float(),
reconstructions)
            optimizer.zero_grad()
            loss_scaler(loss, optimizer, clip_grad=None,
parameters=capsule_net.parameters(), create_graph=second_order)
            targetTensor[batch_id * batch_size:(batch_id + 1) * batch_size, :] =
target
            maskedTensor[batch_id * batch_size:(batch_id + 1) * batch_size, :] =
masked
            train_loss += loss.item()
            if(np.isnan(loss.item())):
                print("Log")
            correctExamplesNum = sum(np.argmax(maskedTensor.data.cpu().numpy(),
1) == np.argmax(targetTensor.data.cpu().numpy(), 1))
        print("Train accuracy:", correctExamplesNum / 132)
        print("Train Loss:", train_loss)

        test_loss = 0
        targetTensorForTest = torch.tensor(np.zeros((33, 15)))
        maskedTensorForTest = torch.tensor(np.zeros((33, 15)))
        with torch.no_grad():
            for batch_id, (data, target) in enumerate(yale.test_loader):

                data, target = Variable(data), Variable(target)
                target = target.view(target.shape[0], target.shape[2])
                if USE_CUDA:
                    data, target = data.cuda(), target.cuda()
                with amp_autocast():
                    output, reconstructions, masked = capsule_net(data.float())
                    loss = capsule_net.loss(data, output, target.float(),
reconstructions)
                targetTensorForTest[batch_id * batchSizeTest:(batch_id + 1) *
batchSizeTest, :] = target
                maskedTensorForTest[batch_id * batchSizeTest:(batch_id + 1) *
batchSizeTest, :] = masked
                test_loss += loss.item()
            correctExamplesNumTest =
sum(np.argmax(maskedTensorForTest.data.cpu().numpy(), 1) ==

np.argmax(targetTensorForTest.data.cpu().numpy(), 1))
            acc = correctExamplesNumTest / 33
        print("Test accuracy:", acc)
        print("Test Loss:", test_loss)
        if(acc > finalAcc[cv]):
            finalAcc[cv] = acc
print(finalAcc)
print("Acc = ", np.mean(finalAcc))
```

Elde ettiğimiz sonuçların grafikleri:



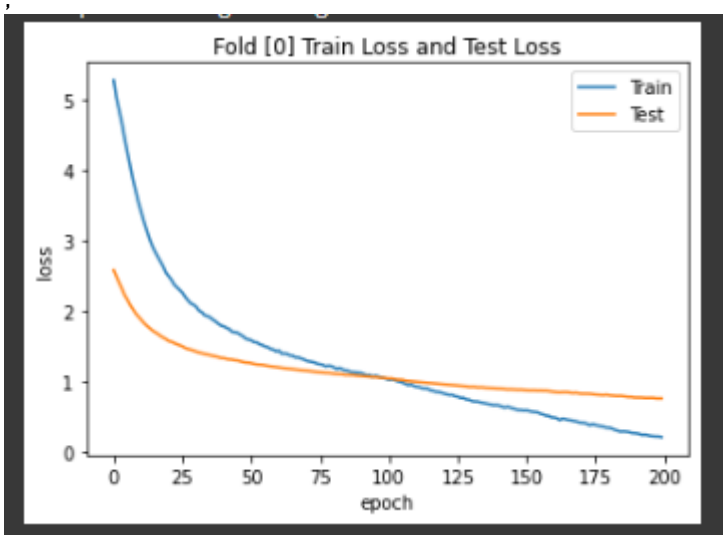


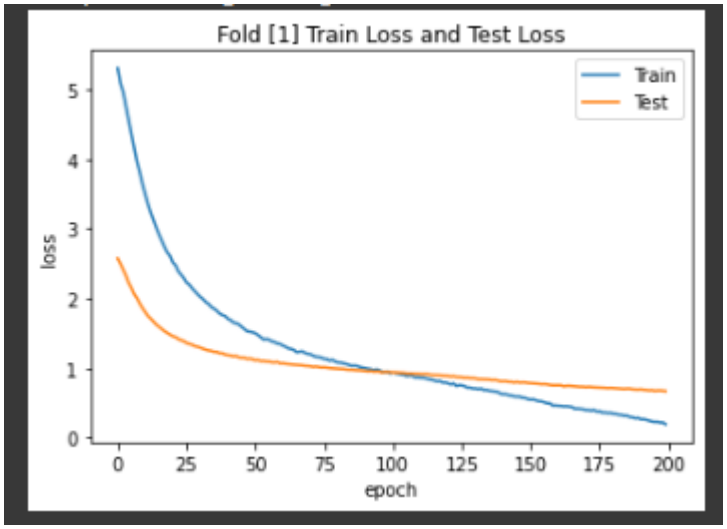


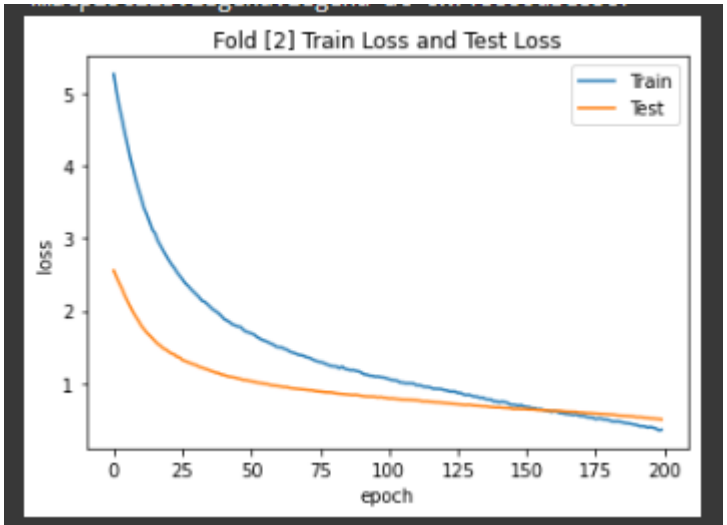

Fold [3] Train Accuracy 200 Epoch
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0
25
50
75
100
125
150
175
200

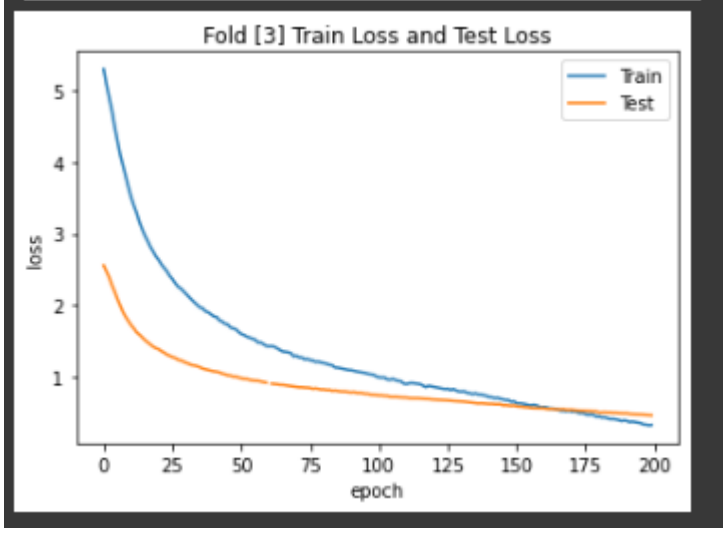
Fold [4] Train Accuracy 200 Epoch
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0
25
50
75
100
125
150
175
200

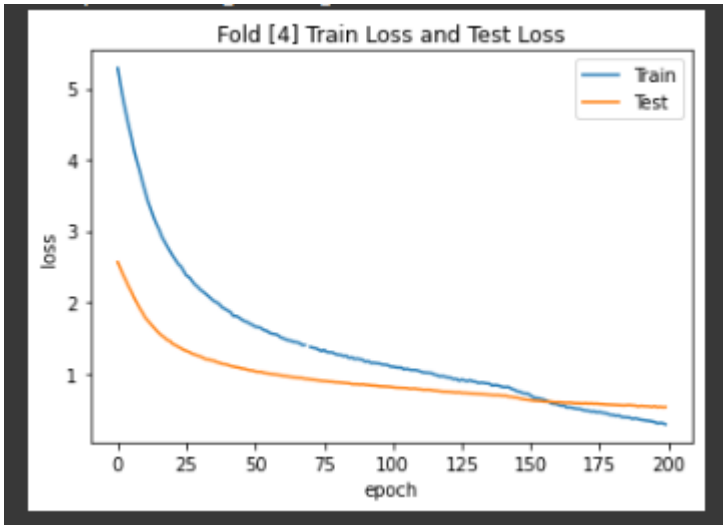
,

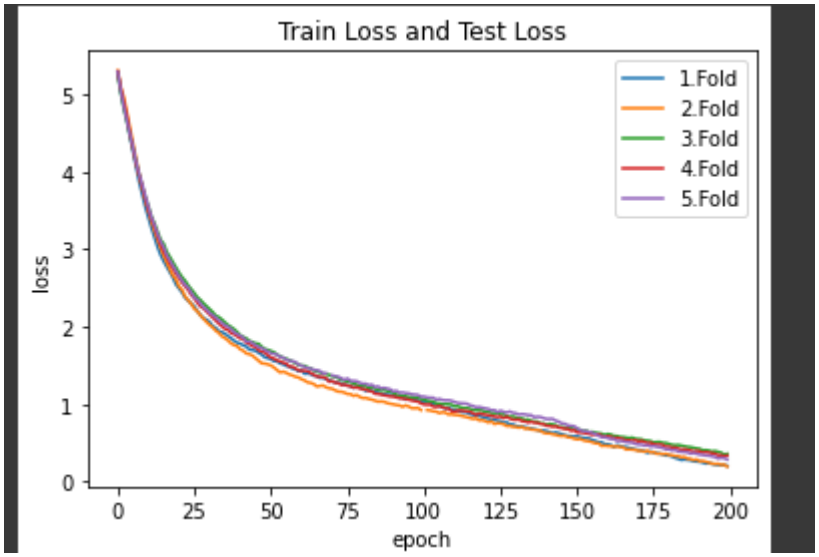

Fold [0] Train Loss and Test Loss
Train
Test
loss
epoch
0
1
2
3
4
5
0
25
50
75
100
125
150
175
200

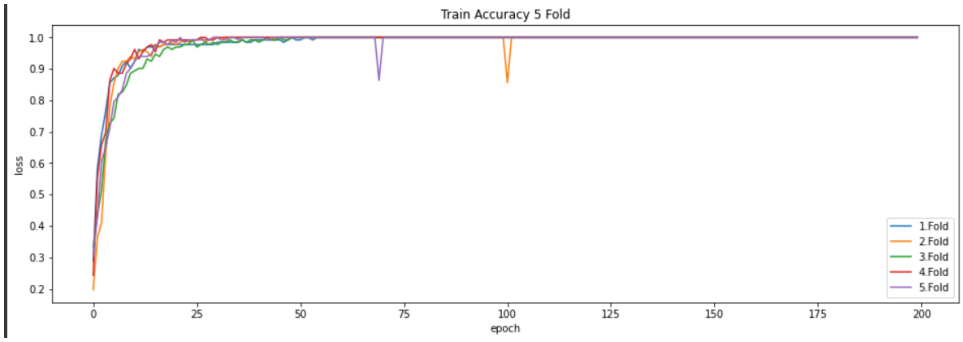

Fold [1] Train Loss and Test Loss
Train
Test
loss
epoch
0
25
50
75
100
125
150
175
200
0
1
2
3
4
5

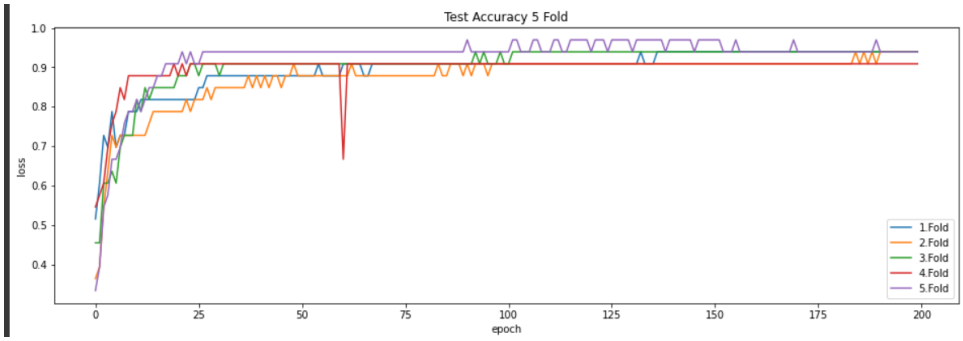

Fold [2] Train Loss and Test Loss
Train
Test
loss
epoch
0
25
50
75
100
125
150
175
200
1
2
3
4
5


Fold [3] Train Loss and Test Loss
Train
Test
loss
epoch
0
25
50
75
100
125
150
175
200
1
2
3
4
5


Fold [4] Train Loss and Test Loss
Train
Test
loss
epoch
0
25
50
75
100
125
150
175
200
1
2
3
4
5


Train Loss and Test Loss
1.Fold
2.Fold
3.Fold
4.Fold
5.Fold
loss
epoch
0
25
50
75
100
125
150
175
200
1
2
3
4
5


Train Accuracy 5 Fold
loss
epoch
1.Fold
2.Fold
3.Fold
4.Fold
5.Fold



Sonuçları detaylı bir şekilde aşağıdaki linkler üzerinden inceleyebilirsiniz.

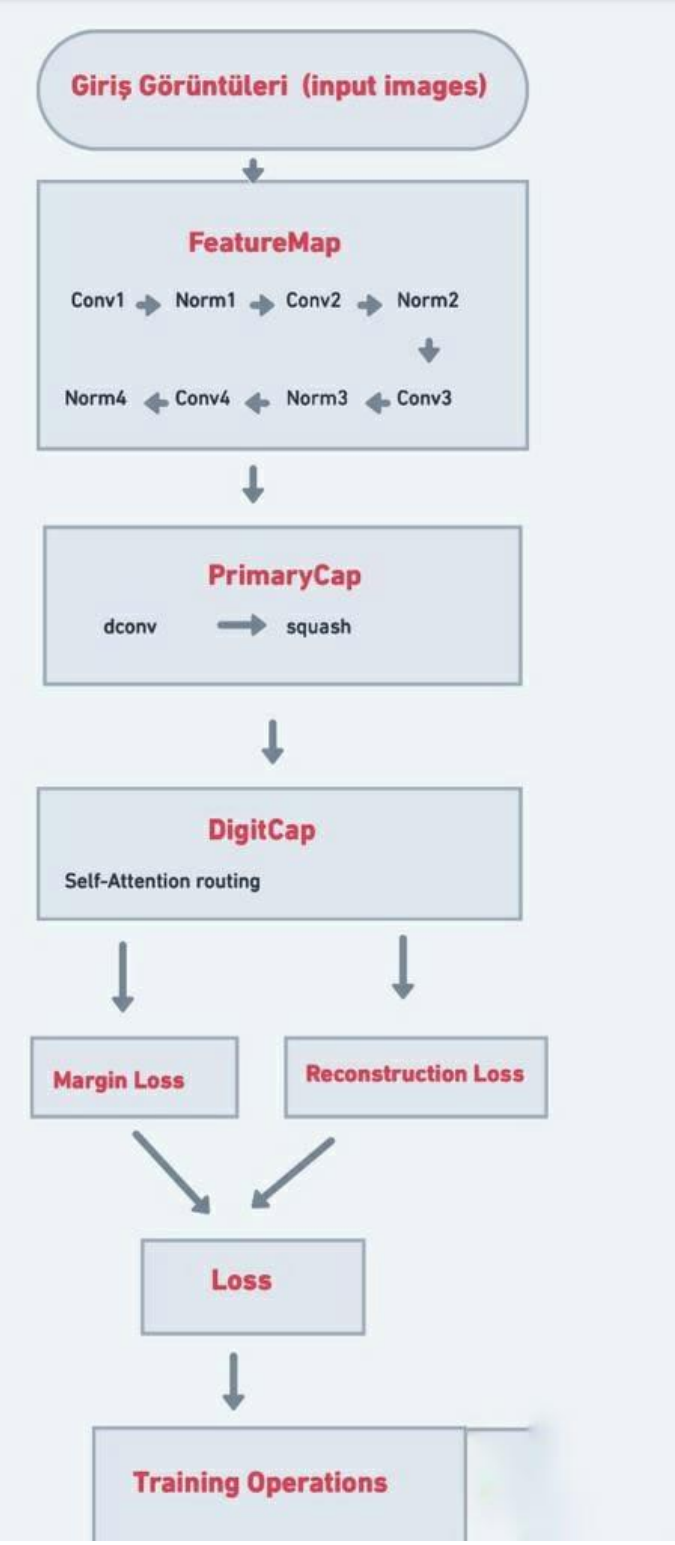
https://colab.research.google.com/drive/1FzkPaYascaNXAQm9mKAYJ2kZVSLyL9FZ?usp=sharing
https://colab.research.google.com/drive/1CRupxmFJInUUSpNDRSQHYiHxwwS0kP4f?usp=sharing

```
[0.93939394 0.96969697 0.93939394 0.90909091 0.93939394]

Acc =  0.9393939393939394
```

5 foldun ortalamasından doğruluk oranını %93 olarak bulmuş bulmaktayız.

2.2.Efficient-Capsnetin Yale VeriSeti Üzerinde Uygulanması

Genel mimari yapısı



Şekil 22: Efficient-Capsnet Mimari yapısı

İlk etapta modelimiz için gerekli parametre değerlerini içeren, bu değerlerin setlenme işlerinin gerçekleştirdiği işlemleri params.py adlı dosyamıza aşağıdaki gibi yazdık.

```
import os
```

```
class CapsNetParam(object):


    __slots__ = [
        "input_width",
        "input_height",
        "input_channel",
        "conv1_filter",
        "conv1_kernel",
        "conv1_stride",
        "conv2_filter",
        "conv2_kernel",
        "conv2_stride",
        "conv3_filter",
        "conv3_kernel",
        "conv3_stride",
        "conv4_filter",
        "conv4_kernel",
        "conv4_stride",
        "dconv_filter",
        "dconv_kernel",
        "dconv_stride",
        "num_primary_caps",
        "dim_primary_caps",
        "num_digit_caps",
        "dim_digit_caps",
    ]

    def __init__(self,
                 input_width = 28,
                 input_height = 28,
                 input_channel= 1,
                 conv1_filter: int = 32,
                 conv1_kernel: int = 5,
                 conv1_stride: int = 1,
                 conv2_filter: int = 64,
                 conv2_kernel: int = 3,
                 conv2_stride: int = 1,
                 conv3_filter: int = 64,
                 conv3_kernel: int = 3,
                 conv3_stride: int = 1,
                 conv4_filter: int = 128,
                 conv4_kernel: int = 3,
                 conv4_stride: int = 2,
                 dconv_kernel: int = 9,
           dconv_stride: int = 1,
                 num_primary_caps: int = 16,
                 dim_primary_caps: int = 8,
                 num_digit_caps: int = 15,
                 dim_digit_caps: int = 16,
                 *args,
                 **kwargs) -> None:

    # Input Specification
        self.input_width = input_width
        self.input_height = input_height
        self.input_channel = input_channel
```

```
        # FeatureMap Layer
        self.conv1_filter = conv1_filter
        self.conv1_kernel = conv1_kernel
        self.conv1_stride = conv1_stride
        self.conv2_filter = conv2_filter
        self.conv2_kernel = conv2_kernel
        self.conv2_stride = conv2_stride
        self.conv3_filter = conv3_filter
        self.conv3_kernel = conv3_kernel
        self.conv3_stride = conv3_stride
        self.conv4_filter = conv4_filter
        self.conv4_kernel = conv4_kernel
        self.conv4_stride = conv4_stride

        # PrimaryCap Layer
        self.dconv_filter = num_primary_caps * dim_primary_caps
        self.dconv_kernel = dconv_kernel
        self.dconv_stride = dconv_stride
        self.num_primary_caps = num_primary_caps
        self.dim_primary_caps = dim_primary_caps

        # DigitCap Layer
        self.num_digit_caps = num_digit_caps
        self.dim_digit_caps = dim_digit_caps

    def get_config(self) -> dict:
        return {
            "input_width": self.input_width,
            "input_height": self.input_height,
            "input_channel": self.input_channel,
            "conv1_filter": self.conv1_filter,
            "conv1_kernel": self.conv1_kernel,
            "conv1_stride": self.conv1_stride,
            "conv2_filter": self.conv2_filter,
            "conv2_kernel": self.conv2_kernel,
            "conv2_stride": self.conv2_stride,
            "conv3_filter": self.conv3_filter,
            "conv3_kernel": self.conv3_kernel,
            "conv3_stride": self.conv3_stride,
            "conv4_filter": self.conv4_filter,
            "conv4_kernel": self.conv4_kernel,
            "conv4_stride": self.conv4_stride,
       "dconv_filter": self.dconv_filter,
            "dconv_kernel": self.dconv_kernel,
            "dconv_stride": self.dconv_stride,
            "num_primary_caps": self.num_primary_caps,
            "dim_primary_caps": self.dim_primary_caps,
            "num_digit_caps": self.num_digit_caps,
        "dim_digit_caps": self.dim_digit_caps
       }

    def save_config(self, path: str) -> None:
                if not isinstance(path, str):
            raise TypeError()
        elif len(path) == 0:
            raise ValueError()
        else:
```

```
            with open(path, 'w', encoding='utf8') as f:
                for k, v in self.get_config().items():
                    f.writelines(f"{k}={v}\n")


def load_config(path: str) -> CapsNetParam:
       if not isinstance(path, str):
        raise TypeError()
    elif len(path) == 0:
        raise ValueError()
    elif not os.path.isfile(path):
        raise FileNotFoundError()

    with open(path, 'r', encoding="utf8") as f:
        config = []
        for l in f.readlines():
            k, v = l.strip().split('=')
            config.append((k, int(v)))
        return CapsNetParam(**dict(config))


def make_param(image_width: int = 28,
               image_height: int = 28,
               image_channel: int = 1,
               conv1_filter: int = 32,
               conv1_kernel: int = 5,
               conv1_stride: int = 1,
               conv2_filter: int = 64,
               conv2_kernel: int = 3,
               conv2_stride: int = 1,
               conv3_filter: int = 64,
               conv3_kernel: int = 3,
               conv3_stride: int = 1,
               conv4_filter: int = 128,
               conv4_kernel: int = 3,
               conv4_stride: int = 2,
               dconv_kernel: int = 9,
               dconv_stride: int = 1,
               num_primary_caps: int = 16,
               dim_primary_caps: int = 8,
               num_digit_caps: int = 10,
                  dim_digit_caps: int = 16) -> CapsNetParam:
    return CapsNetParam(
        image_width,
        image_height,
        image_channel,
        conv1_filter,
        conv1_kernel,
         conv1_stride,
    conv2_filter,
        conv2_kernel,
        conv2_stride,
        conv3_filter,
        conv3_kernel,
        conv3_stride,
        conv4_filter,
        conv4_kernel,
        conv4_stride,
```

```
        dconv_kernel,
        dconv_stride,
        num_primary_caps,
        dim_primary_caps,
        num_digit_caps,
        dim_digit_caps,
    )
```

Daha sonra layer.py adlı bir dosyaya, katmanların yazım işlevini gerçekleştirdik.
Bu dosyada ilk etapta kullanacağımız squash işlevini gerçekleştirecek Squash sınıfının tanımlanmasını yaptık.

```
class Squash(nn.Module):
    def __init__(self, eps=1e-7, name="squash"):
        super().__init__()
        self.eps = eps

    def forward(self, input_vector):
        norm = torch.norm(input_vector, dim=-1, keepdim=True)
        coef = 1 - 1 / torch.exp(norm)
        unit = input_vector / (norm + self.eps)
        return coef * unit

    def compute_output_shape(self, input_shape):
        return input_shape
```

Öncelikle Batch normalizyasyon ne demek?

Eğitim esnasında her katman hatasını düzeltmeye çalışır. Fakat katmanlar bağımsız hareket etmekte. Mesela 1. katman çıktısı ile 2. katman beslenir bu nedenle bir katmandaki değişim diğer katmanları da etkileyecektir**.** Giriş katmanlarındaki bu kaymalar nedeniyle hesaplamalar için zaman gittikçe artar. (3.katman'daki öğrenmeye başlamak için 2.katmanın öğrenmesinin bitmesi gerekiyor çünkü) Bu kaymaları azaltmak için batch normalization uygulanır. Genel olarak normalizasyon işlevi tüm girdilerin dağılımının ortalamasını 0, standart sapmasını 1 olacak hale getirmektedir. Dolayısıyla -1 ile 1 arasına sıkıştırılmış olur. Bu normalizasyon işlemini girişe uygulayabiliriz ancak bu normalleştirmeden 2. katman faydalanamaz. Bu nedenle katmanlar arasında normalizasyon yaparız. Batch normalizayon burada devreye giriyor.
Batch normalization sayesinde ağdaki katmanlar, önceki katmanın öğrenmesini beklemek zorunda kalmaz. Eş zamanlı olarak öğrenime olanak sağlar. Eğitimimizin hızlanmasını sağlar. Orjinal capsnetten asıl ayrıldığı kısım burası olmakta. Batch norm ağın daha kararlı ve düzenli olmasına da olanak sağlar.

FeatureMap
Giriş katmanı modelin ilk katmanıdır. Giriş olarak alınan bu veriler evrişim katmanına gönderilir. Öncelikle girişlerimiz bu FeatureMap katmanına gönderilir ve bu katman evrişimli sinir ağında bulunan evrişim katmanlarının aynısıdır. Bu katmanda gelen giriş 4 tane

konvülsiyon işleminden geçer. Her konvülsiyon sonucu Batch normalizasyon fonksiyonuna gönderilerek norm 1,2,3 ve 4 sonuçları elde edilir.

İlk girdi torch.Size([22, 1, 28, 28])
22 batch size, 1 ise gri tonlamada görüntüleri, 28 28 görüntü boyutlarını ifade etmektedir.

Conv1 sonucunun batch normalizasyon fonksiyonuna gönderilerek elde edilen norm1 sonucu ise:
torch.Size([22, 56, 24, 24])
Bu conv1 işleminde kernel_size 5, out_channels 56 değerlerine setlenmiştir.
Görüntü verilerinin boyutu (kernel_size-1) kadar piksel kaybeder. Kaydırma adımı 1 olduğundan 5-1=4 tane piksel kaybetmesi sonucu 24 değeri elde edilir. Norm1 çıkış değerlerindeki ilk parametre 22 batch_size, 56 out_channels, 24 ise giriş görüntülerinin piksel kaybetmesi sonucu elde edilmiş görüntü boyutudur.

Conv2 sonucunun batch normalizasyon fonksiyonuna gönderilerek elde edilen norm2 sonucu ise:
torch.Size([22, 128, 22, 22])
Yukarıdaki aynı adımların burada da uygulanması sonucunda değerler elde edilmiştir. Burada kernel_size değerimiz 3 ve stride değeri 1 olduğundan görüntümüz 2 piksel kaybederek 22 22 görüntü boyutları elde edilmiştir. 128 değeri ise out_channel değerimizi ifade etmektedir.

Conv3 sonucunun batch normalizasyon fonksiyonuna gönderilerek elde edilen norm3 sonucu ise:
torch.Size([22, 128, 20, 20])
Aynı işlemler tekrar edilir. Burada da kernel_size3 stride değerimiz 1 olduğundan görüntü boyutumuz 2 piksel değer kaybederek 20 20 görüntü boyutları elde edilmiştir.
56 değeri ise out_channel değerimizi ifade etmektedir.

Conv4 sonucunun batch normalizasyon fonksiyonuna gönderilerek elde edilen norm4 sonucu ise:
torch.Size([22, 256, 9, 9])
Burada 20x20 boyutlarına indirgenen görüntünün kaydırma adımı 2 olan parametresiyle boyutları 10x10 boyutlarına düşürülmüştür. Bununla beraber çekirdek boyutu 3 alınarak (çekirdek boyutu-1=2/2=1) görüntü 1 piksel daha kaybedip bu katman çıkışında 9x9 boyutuna gelmiştir.

Bu conv4 sonucu ise featureMap katmanının çıkış katmanı olarak bir sonraki katmana giriş olarak verilir.

```
class FeatureMap(nn.Module):
   def __init__(self, param: CapsNetParam) -> None:
        super(FeatureMap, self).__init__()
        self.param = param
        self.conv1 = nn.Sequential(nn.Conv2d(
            in_channels = 1,
            out_channels = self.param.conv1_filter,
            kernel_size = self.param.conv1_kernel,
            stride = self.param.conv1_stride),
```

```
            nn.ReLU())
        self.norm1 = nn.BatchNorm2d(self.param.conv1_filter)

     self.conv2 = nn.Sequential(nn.Conv2d(
            in_channels = self.param.conv1_filter,
            out_channels = self.param.conv2_filter,
            kernel_size = self.param.conv2_kernel,
            stride = self.param.conv2_stride),
            nn.ReLU())
        self.norm2 = nn.BatchNorm2d(self.param.conv2_filter)

        self.conv3 = nn.Sequential(nn.Conv2d(
            in_channels= self.param.conv2_filter,
            out_channels = self.param.conv3_filter,
            kernel_size = self.param.conv3_kernel,
            stride = self.param.conv3_stride),
            nn.ReLU())
        self.norm3 = nn.BatchNorm2d(self.param.conv3_filter)

        self.conv4 = nn.Sequential(nn.Conv2d(
            in_channels = self.param.conv3_filter,
            out_channels = self.param.conv4_filter,
            kernel_size = self.param.conv4_kernel,
            stride = self.param.conv4_stride),
            nn.ReLU())
        self.norm4 = nn.BatchNorm2d(self.param.conv4_filter)


   def forward(self, input_images: torch.Tensor) -> torch.Tensor:
       feature_maps = self.norm1(self.conv1(input_images))
       feature_maps = self.norm2(self.conv2(feature_maps))
       feature_maps = self.norm3(self.conv3(feature_maps))
       return self.norm4(self.conv4(feature_maps))
```

PrimaryCap

Burada FeatureMap katmanını çıkış değeri girdi olarak bu katmana verilir.
torch.Size([22, 256, 9, 9]).
Bu değer Squash fonksiyonuna gönderilerek **torch.Size([22, 16, 8])** çıktısı elde edilir.
22 batch_size, 16 num_primary_caps değerini ifade ederken 8 ise dim_primary_caps değerini ifade etmektedir.

Burada dconv_filter değeri **self.dconv_filter = num_primary_caps * dim_primary_caps** şeklinde param.py dosyasında tanımlanmıştır. Bu çıkış değerleri de bir sonraki katman olan DigitCap katmanına gönderilir.

```
class PrimaryCap(nn.Module):
    def __init__(self, param: CapsNetParam) -> None:
        super(PrimaryCap, self).__init__()
        self.param = param
        self.dconv = nn.Sequential(nn.Conv2d(
```

```
            in_channels = self.param.conv4_filter,
            out_channels = self.param.dconv_filter,
            kernel_size = self.param.dconv_kernel,
            stride = self.param.dconv_stride,
            groups = self.param.dconv_filter),
            nn.ReLU())
        self.squash = Squash()

    def forward(self, feature_maps):
        dconv_outputs = self.dconv(feature_maps)
        return self.squash(dconv_outputs.view(dconv_outputs.shape[0], -1,
self.param.dim_primary_caps))
```

DigitCap

Burada Self attention algoritması kullanılmıştır. Self attention görüntüdeki herhangi bir özelliğin diğer özelliklerle ilişkisini ortaya çıkaran bir mekanizma olarak düşünülebilir.
Bu katmana giriş olarak PrimaryCap katmanının çıkış değerleri verilmektedir.
torch.Size([22, 16, 8]).
Bu giriş değerleri ilk etapta unsqueeze işlemi uygulanır. Burada istenilen konumda yeni bir boyut ekleyerek çıktı olarak yeni bir tensör oluşturma işlemi için kullanılır.
Daha sonra elde ettiğimiz tensör üzerimde squeze işlemi kullanılr. Squeze işlemi ise 1 boyutuna sahip olan tensör düşürülür. Ve bu adımlardan sonra Softmax sınıflandırıcısı kullanılır.
Softmax her bir görüntü için giriş değerlerini kullanarak çıktıyı belirleyecek olan olasılık değerlerinin hesaplanması işlemini gerçekleştirilir.

Sonuç çıktı değerimiz:
torch.Size([22, 15, 16])
22 batc_size, 15 değeri num_digit_caps 16 değerini ifade eder.

```
class DigitCap(nn.Module):
    def __init__(self, param: CapsNetParam) -> None:
        super(DigitCap, self).__init__()
        self.param = param
        self.attention_coef = 1 /
torch.sqrt(torch.tensor(self.param.dim_primary_caps))
        self.W = nn.Parameter(torch.randn(self.param.num_digit_caps,
self.param.num_primary_caps,
                self.param.dim_digit_caps, self.param.dim_primary_caps))
        self.B = nn.Parameter(torch.randn(self.param.num_digit_caps, 1,
self.param.num_primary_caps))
        self.squash = Squash()
        self.softmaxFunc = nn.Softmax(dim = -2)

    def forward(self, primary_caps):
        U = torch.unsqueeze(torch.tile(
                            torch.unsqueeze(primary_caps, axis = 1), [1,
self.param.num_digit_caps, 1, 1]),
                            axis = -1)
```

```
        U_hat = torch.squeeze(torch.matmul(self.W, U), axis = -1)
        A = self.attention_coef * torch.matmul(U_hat, U_hat.transpose(2, 3))
        C = self.softmaxFunc(torch.sum(A, axis = -2, keepdims = True))
        S = torch.squeeze(torch.matmul(C + self.B, U_hat), axis=-2)
        return self.squash(S)
```

Ilk etapta yale veri seti üzerinde ön işlemler gerçekleştirilir. Cross-validation işlemi uygulanır.
Layer.py olusturulduktan sonra ise modeli aşağıdaki kod blokları ile oluşturup loss değerlerinin hesaplanması ile training operasyonlar gerçekleştirmiştir.
Tahmin çıktısı ile sağlanan hedef değer arasındaki hatayı ölçmek için kayıp fonksiyonları kullanılır. Bir kayıp fonksiyonu bize algoritma modelinin beklenen sonucu gerçekleştirmekten ne kadar uzak olduğunu söyleyen yapıdır.

Margin Loss ve reconstruction Başarım İyileştirme Yöntemleri (Regularization)

Başarım iyileştirme yöntemlerini modelden bağımsız, modelin başarımını artıran harici bir parametre olarak kullanılmaktadır. Regularization teknikleri genelde tasarlanan modelde overfitting önleyerek başarımı artırmak için kullanılmaktadır. Buna ek olarak başarımı düşürmeden modelin karmaşıklığını (complexity) azaltmak için de kullanıldığı durumlar vardır.
Pytorch sonraki geçişler için gradyanları biriktirdiğinden dolayı her epochda optimizer.zero_grad() kullanılarak gradyanları sıfırlarız.Daha sonra eğitim örneklerinden yeniden yapılandırma kaybını hesaplarız ve train_loss.backward() fonksiyonu kullanılarak hesaplanan mevcut gradyanlara dayalı olarak optimizer.step() ile modelimizin optimize işlemini gerçekleştirmiş oluruz. Eğitimimizin nasıl gittiğini görmek için, her epoch için eğitim kaybını toplarız.train_loss + = loss.item() değerini hesaplarız.

```
import torch.utils.data as torchdata
from pathlib import Path
import cv2
import glob
from timm.utils import AverageMeter, CheckpointSaver, NativeScaler
from sklearn.metrics import accuracy_score
from PIL import Image
import warnings
from torchvision import transforms, utils
from torch.utils.data import Dataset, DataLoader
import matplotlib.pyplot as plt
from skimage import io, transform
import pandas as pd
import os
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
from torch.optim import Adam
from torchvision import datasets, transforms
import torch.optim as optim
```

```
from layers import DigitCap
from layers import FeatureMap
from layers import PrimaryCap
from losses import MarginLoss
from param import CapsNetParam
USE_CUDA = False

warnings.filterwarnings("ignore")
plt.ion()

class TrainDataset(torchdata.Dataset):
    def __init__(self, fileList, labels):
        self.fileList = fileList
        self.labels = labels

    def __len__(self):
        return len(self.fileList)

    def __getitem__(self, index: int):
        filename = self.fileList[index]
        img = Image.open(filename)
        param = CapsNetParam()
        img = np.array(img.resize((param.input_width, param.input_height),
Image.ANTIALIAS))
        # img = img / 255  ## normalize
        dataset_transform = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize((0.1307,), (0.3081,))
        ])
        img = dataset_transform(img)
        label = torch.sparse.torch.eye(15).index_select(
            dim=0, index=torch.tensor(self.labels[index]))

        return img, label


class Yale:
    def __init__(self, _batch_size, batchSizeTest, trainIndex, testIndex):
        train_dataset =
TrainDataset(np.array(fileLinks)[trainIndex.astype(int)], labels[trainIndex])
        test_dataset = TrainDataset(np.array(fileLinks)[testIndex.astype(int)],
labels[testIndex])
        self.train_loader = torch.utils.data.DataLoader(train_dataset,
batch_size = _batch_size, shuffle=True)
        self.test_loader = torch.utils.data.DataLoader(test_dataset, batch_size
= batchSizeTest, shuffle=False)

def build_dataset():
    fileList = []
    labels = []
    for i in range(1, 16):
        files = glob.glob('./data/subject'+str(i).zfill(2)+"*")
        for fname in files:
            fileList.append(fname)
            labels.append(i - 1)
    return fileList, np.array(labels)
```

```
class CapsNet(nn.Module):

    def __init__(self):
        super(CapsNet, self).__init__()
        param = CapsNetParam()
        self.conv_layer = FeatureMap(param)
        self.primary_capsules = PrimaryCap(param)
        self.digit_capsules = DigitCap(param)
        self.mse_loss = nn.MSELoss()

    def forward(self, data):
        output =
self.digit_capsules(self.primary_capsules(self.conv_layer(data)))
        digit_probs = torch.norm(output, dim = -1)
        return digit_probs

    def loss(self, data, x, target, reconstructions):
        return self.margin_loss(x, target) + self.reconstruction_loss(data,
reconstructions)

    def margin_loss(self, x, labels, size_average=True):
        batch_size = x.size(0)

        v_c = torch.sqrt((x**2).sum(dim=2, keepdim=True))

        left = F.relu(0.9 - v_c).view(batch_size, -1)
        right = F.relu(v_c - 0.1).view(batch_size, -1)

        loss = labels * left + 0.5 * (1.0 - labels) * right
        loss = loss.sum(dim = 1).mean()

        return loss

    def reconstruction_loss(self, data, reconstructions):
        loss = self.mse_loss(reconstructions.view(
            reconstructions.size(0), -1), data.view(reconstructions.size(0), -
1))
        return loss * 0.0005

batch_size = 22
batchSizeTest = int(batch_size / 2)
n_epochs = 200
finalAcc = np.zeros(5)
fileLinks, labels = build_dataset()
from sklearn.model_selection import StratifiedKFold
skf = StratifiedKFold(n_splits = 5)
for cv, (train_index, test_index) in enumerate(skf.split(fileLinks, labels)):

    yale = Yale(batch_size, batchSizeTest, train_index, test_index)
    import model
    capsule_net = CapsNet()
    if USE_CUDA:
        capsule_net = capsule_net.cuda()
    optimizer = optim.Adamax(capsule_net.parameters(), lr=0.0001)

    amp_autocast = torch.cuda.amp.autocast
    loss_scaler = NativeScaler()
    lossFunction = torch.nn.MSELoss()
```

```
    second_order = hasattr(optimizer, 'is_second_order') and
optimizer.is_second_order
    for epoch in range(n_epochs):

        print("Fold = ", str(cv), "Epoch = ", str(epoch + 1))
        capsule_net.train()
        train_loss = 0
        targetTensor = torch.tensor(np.zeros((132, 15)))
        maskedTensor = torch.tensor(np.zeros((132, 15)))
        for batch_id, (data, target) in enumerate(yale.train_loader):

            data, target = Variable(data), Variable(target)
            target = target.view(target.shape[0], target.shape[2])
            if USE_CUDA:
                data, target = data.cuda(), target.cuda()
            with amp_autocast():
                output = capsule_net(data.float())
                loss = lossFunction(output, target.float())
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            targetTensor[batch_id * batch_size:(batch_id + 1) * batch_size, :] =
target
            maskedTensor[batch_id * batch_size:(batch_id + 1) * batch_size, :] =
output
            train_loss += loss.item()
            if(np.isnan(loss.item())):
                print("Log")
            correctExamplesNum = sum(np.argmax(maskedTensor.data.cpu().numpy(),
1) == np.argmax(targetTensor.data.cpu().numpy(), 1))
        print("Train accuracy:", correctExamplesNum / 132)
        print("Train Loss:", train_loss)

        test_loss = 0
        targetTensorForTest = torch.tensor(np.zeros((33, 15)))
        maskedTensorForTest = torch.tensor(np.zeros((33, 15)))
        with torch.no_grad():
            for batch_id, (data, target) in enumerate(yale.test_loader):

                data, target = Variable(data), Variable(target)
                target = target.view(target.shape[0], target.shape[2])
                if USE_CUDA:
                    data, target = data.cuda(), target.cuda()
                with amp_autocast():
                    output = capsule_net(data.float())
                    loss = lossFunction(output, target.float())
                targetTensorForTest[batch_id * batchSizeTest:(batch_id + 1) *
batchSizeTest, :] = target
                maskedTensorForTest[batch_id * batchSizeTest:(batch_id + 1) *
batchSizeTest, :] = output
                test_loss += loss.item()
            correctExamplesNumTest =
sum(np.argmax(maskedTensorForTest.data.cpu().numpy(), 1) ==

np.argmax(targetTensorForTest.data.cpu().numpy(), 1))
            acc = correctExamplesNumTest / 33
        print("Test accuracy:", acc)
        print("Test Loss:", test_loss)
        if(acc > finalAcc[cv]):
```

```
            finalAcc[cv] = acc
print(finalAcc)
print("Acc = ", np.mean(finalAcc))
```

Kullandığımız farklı hipermetre değerleri ile ne yazık ki elde ettiğimiz en iyi sonuç %18'dir ve bu da beklenenden daha kötü bir değer.
Kullanıdığımız bazı hipermetre ayarları ve elde ettiğimiz sonuçlar aşağıda verilmiştir.

1. optimizer = optim.AdamW(capsule_net.parameters(), lr=0.0001)
batch_size=22
[0.15151515 0.09090909 0.15151515 0.24242424 0.12121212]
Acc = 0.15151515151515152

2. optimizer = optim.AdamW(capsule_net.parameters(), lr=0.0001)
batch_size =6

[0.15151515 0.21212121 0.12121212 0.15151515 0.15151515]
Acc = 0.15757575757575756

3. optimizer = optim.AdamW(capsule_net.parameters(), lr=0.001)
batch_size=22
[0.15151515 0.12121212 0.12121212 0.12121212 0.15151515]
Acc = 0.13333333333333333

4.optimizer = optim.AdamW(capsule_net.parameters(), lr=0.01)
batch_size=22
[0.15151515 0.12121212 0.12121212 0.12121212 0.12121212]
Acc = 0.12727272727272726

5. optimizer = optim.Adamax(capsule_net.parameters(), lr=0.0001)
batch_size=22
[0.21212121 0.18181818 0.21212121 0.18181818 0.15151515]
Acc = 0.18787878787878787

6.optimizer = optim.Adamax(capsule_net.parameters(), lr=0.01)
batch_size=22
[0.15151515 0.18181818 0.12121212 0.15151515 0.18181818]
Acc = 0.15757575757575756

7. optimizer = optim.SGD(capsule_net.parameters(), lr=0.01,momentum=0.9)
batch_size=22
[0.24242424 0.12121212 0.21212121 0.15151515 0.12121212]
Acc = 0.1696969696969697

8.optimizer = optim.RMSprop(capsule_net.parameters(), lr=0.0001, momentum=0.9)
batch_size=32

[0.18181818 0.15151515 0.15151515 0.15151515 0.15151515]
Acc = 0.15757575757575756

9.optimizer = optim.Adagrad(capsule_net.parameters(), lr=0.001)
batch_size = 64
[0.18181818 0.15151515 0.18181818 0.21212121 0.15151515]
Acc = 0.17575757575757575

10 .optimizer = optim.Adagrad(capsule_net.parameters(), lr=0.01)
batch_size = 66
[0.18181818 0.12121212 0.12121212 0.12121212 0.15151515]
Acc = 0.13939393939393938

2.3 Dense Capsule Network'ün Yale VeriSeti Üzerinde Uygulanması

Bu çalışmamızda yale veri setini kullanarak DcNet yöntemini pytorch framework'ü ile gerçeklenmesini ele alacağız.

```
import torch.nn.functional as F
from torch.autograd import Variable
from torchvision import datasets, transforms
from capsnet import CapsNet
from data_loader import Dataset
from tqdm import tqdm

import glob
import data_loader as myDL
from timm.utils import NativeScaler
from sklearn.model_selection import StratifiedKFold
```

Gerekli kütüphanelerin import edilmesi;

Gerekli kütüphaneleri ekledikten sonra girdilerin birbiriyle aynı aralıktaki sayılardan oluşturmak için ölçeklendirme ve normaliz işlemlerini yaparız.

Class yale sınıfı içerisinde hazır hale getiriyoruz.

```
import glob
def build_dataset():
    fileList = []
    labels = []
    for i in range(1, 16):
        files = glob.glob('./data/subject'+str(i).zfill(2)+"*")
        for fname in files:
            fileList.append(fname)
            labels.append(i - 1)
    return fileList, np.array(labels)

import data_loader as myDL
class Yale:
    def __init__(self, _batch_size, batchSizeTest, trainIndex, testIndex):
        train_dataset =
myDL.TrainDataset(np.array(fileLinks)[trainIndex.astype(int)],
labels[trainIndex])
        test_dataset =
myDL.TrainDataset(np.array(fileLinks)[testIndex.astype(int)],
labels[testIndex])
        self.train_loader = torch.utils.data.DataLoader(train_dataset,
batch_size = _batch_size, shuffle=True)
        self.test_loader = torch.utils.data.DataLoader(test_dataset,
batch_size = batchSizeTest, shuffle=False
```

Modelin ilk katmanı olan giriş katmanında alınan verilen evrişim katmanına iletilir. Convlayer bu girişlerin ilk iletildiği yerdir.

```
class ConvLayer(nn.Module):
    def __init__(self, in_channels=1, out_channels=256, kernel_size=9):
        super(ConvLayer, self).__init__()
        self.conv = nn.Conv2d(in_channels=in_channels,
                              out_channels=out_channels,
                              kernel_size=kernel_size,
                              stride=1
                              )
        def forward(self, x):
        return F.relu(self.conv(x))
```

Evrişim katmanından sonra görüntü boyutları 20x20 pikselle sınırlanmıştır. Sonrasın da elde ettiğimiz bu görüntülere özellik kapsülü(primary) uygulanmıştır.

Evrişim katmanından farklı olarak boyutlandırma ve squash fonksiyonları uygulanmıştır. Primary kapsülü'ne uygulanan görüntü boyutları 10x10 boyutuna kadar sınırlandırılmıştır. Beraberinde çekirdek boyutu 9 alınarak 4 piksel daha kaybeden görüntü çıkışta 6x6 boyutuna sınırlanmıştır. Daha sonra görüntüler vektör haline getirilebilmek için yeniden boyutlandırma işlemi işe vektör boyutuna indirgenmiştir.

```
class PrimaryCaps(nn.Module):
    def __init__(self, num_capsules=8, in_channels=256, out_channels=32,
kernel_size=9, num_routes=32 * 6 * 6):
        super(PrimaryCaps, self).__init__()
        self.num_routes = num_routes
        self.capsules = nn.ModuleList([
            nn.Conv2d(in_channels=in_channels, out_channels=out_channels,
kernel_size=kernel_size, stride=2, padding=0)
            for _ in range(num_capsules)])
    def forward(self, x):
        u = [capsule(x) for capsule in self.capsules]
        u = torch.stack(u, dim=1)
        u = u.view(x.size(0), self.num_routes, -1)
        return self.squash(u)
    def squash(self, input_tensor):
        squared_norm = (input_tensor ** 2).sum(-1, keepdim=True)
        output_tensor = squared_norm * input_tensor / ((1. + squared_norm) *
torch.sqrt(squared_norm))
        return output_tensor
```

DigitCaps sınıfında Dinamik yönlendirme algoritması ile birincil kapsüller ile yapılan tahminlere uygun olarak özellik yönelimlerine dayalı sınıflar hakkında tahminde bulunur bu katmandaki kapsüller özellik yönelimine dayalı olarak tahminde bulunur.

```
class DigitCaps(nn.Module):
    def __init__(self, num_capsules=10, num_routes=32 * 6 * 6, in_channels=8,
out_channels=16):
        super(DigitCaps, self).__init__(
        self.in_channels = in_channels
        self.num_routes = num_routes
        self.num_capsules = num_capsules
       self.W = nn.Parameter(torch.randn(1, num_routes, num_capsules,
out_channels,    in_channels))
   def forward(self, x):
        batch_size = x.size(0)
        x = torch.stack([x] * self.num_capsules, dim=2).unsqueeze(4)
        W = torch.cat([self.W] * batch_size, dim=0)
        u_hat = torch.matmul(W, x)
        b_ij = Variable(torch.zeros(1, self.num_routes, self.num_capsules, 1))
        if USE_CUDA:
            b_ij = b_ij.cuda()
        num_iterations = 3
```

```
for iteration in range(num_iterations):
            c_ij = F.softmax(b_ij, dim=1)
            c_ij = torch.cat([c_ij] * batch_size, dim=0).unsqueeze(4)
            s_j = (c_ij * u_hat).sum(dim=1, keepdim=True)
            v_j = self.squash(s_j)
            if iteration < num_iterations - 1:
                a_ij = torch.matmul(u_hat.transpose(3, 4), torch.cat([v_j] *
self.num_routes, dim=1))
                b_ij = b_ij + a_ij.squeeze(4).mean(dim=0, keepdim=True)
        return v_j.squeeze(1)
    def squash(self, input_tensor):
        squared_norm = (input_tensor ** 2).sum(-1, keepdim=True)
        output_tensor = squared_norm * input_tensor / ((1. + squared_norm) *
torch.sqrt(squared_norm))
        return output_tensor
```

Düzenleyici bir görev üstlenen decoder sınıfı digit kapsülünden alınan doğru vektörü etiketli bir görüntüye dönüştürmeyi öğrenir. Digit sınıfının çıktısını girdi olarak alır ve görüntüyü yeniden oluşturmayı amaçlar. Kayıp işlevi çıkış görüntüsü ile giriş görüntüsü arasındaki Öklid uzaklığıdır. Elde edilen görüntü ne kadar yakınsa o kadar başarılıdır.

```
class Decoder(nn.Module):
    def __init__(self, input_width=28, input_height=28, input_channel=1):
        super(Decoder, self).__init__()
        self.input_width = input_width
        self.input_height = input_height
        self.input_channel = input_channel
        self.reconstraction_layers = nn.Sequential(
            nn.Linear(16 * 15, 512),
            nn.ReLU(inplace=True),
            nn.Linear(512, 1024),
            nn.ReLU(inplace=True),
            nn.Linear(1024, self.input_height * self.input_width *
self.input_channel),
            nn.Sigmoid()
        )
       def forward(self, x, data):
        classes = torch.sqrt((x ** 2).sum(2))
        classes = F.softmax(classes, dim=0)

        _, max_length_indices = classes.max(dim=1)
        masked = Variable(torch.sparse.torch.eye(15))
        if USE_CUDA:
            masked = masked.cuda()
        masked = masked.index_select(dim=0,
index=Variable(max_length_indices.squeeze(1).data))
        t = (x * masked[:, :, None, None]).view(x.size(0), -1)
        reconstructions = self.reconstraction_layers(t)
        reconstructions = reconstructions.view(-1, self.input_channel,
self.input_width, self.input_height)
        return reconstructions, masked
```

Eğitim döngüsünde modeldeki veri yükleyiciden alınan örnekler üzerinde değerlendirme yaparız. Kayıp fonksiyonu ile model ve hedef çıktılarını karşılaştırız. Hedefe daha da yakınlaştırmak için modeli yormamız gerekebilir. Eğitimde oluşan kayıp, modelin eğitim setine uyumu hakkında fikir verir. Eğer azalma yoksa model veri için basittir diyebiliriz. Ayrıca verilerimiz anlamlı bilgi içermiyor da olabilir.

Kodun bu kısmı programın işlevsel kısmı denilebilir. Kodun bu parçasında CapsNet class'ı ve

```
class CapsNet(nn.Module):
    def __init__(self, config=None):
        super(CapsNet, self).__init__()
        if config:
            self.conv_layer = ConvLayer(config.cnn_in_channels,
config.cnn_out_channels, config.cnn_kernel_size)
            self.primary_capsules = PrimaryCaps(config.pc_num_capsules,
config.pc_in_channels, config.pc_out_channels,
                                                config.pc_kernel_size,
config.pc_num_routes)
            self.digit_capsules = DigitCaps(config.dc_num_capsules,
config.dc_num_routes, config.dc_in_channels,
                                            config.dc_out_channels)
            self.decoder = Decoder(config.input_width, config.input_height,
config.cnn_in_channels)
        else:
            self.conv_layer = ConvLayer()
            self.primary_capsules = PrimaryCaps()
            self.digit_capsules = DigitCaps()
            self.decoder = Decoder()
            self.mse_loss = nn.MSELoss()
```

```
        return self.margin_loss(x, target) + self.reconstruction_loss(data,
reconstructions)
    def margin_loss(self, x, labels, size_average=True):
        batch_size = x.size(0)
        v_c = torch.sqrt((x ** 2).sum(dim=2, keepdim=True))
        left = F.relu(0.9 - v_c).view(batch_size, -1)
        right = F.relu(v_c - 0.1).view(batch_size, -1)
        loss = labels * left + 0.5 * (1.0 - labels) * right
        loss = loss.sum(dim=1).mean()
        return loss
   def reconstruction_loss(self, data, reconstructions):
        loss = self.mse_loss(reconstructions.view(reconstructions.size(0), -
1), data.view(reconstructions.size(0), -1))
        return loss * 0.0005
```

DCnet'e göre modifiye edilmiş hali bulunmaktadır. DCnet algoritmasının gerekliliklerinden olan ve CapsNet'te de asıl işlemleri gerçekleştiren fonksiyonlar bu CapsNet class'ında yer almaktadır.
Bunlar başlıca ConvLayer , PrimaryCaps , DigitCaps ve Decoder fonksiyonudur. Bu fonksiyonlarla birlikte class'ın açıklamasını yapacak olursak; başlangıçta if-else bloğu ile config adı altında konfigürasyon sağlanır.

Bu kontrol, veri setinin üzerinde yapılacak işlemlere uygun olup olmadığını, uygun olmaması durumda else bloğu ile yapılması gereken işlemlere yönlendirir. İf bloğuna girebilen veri seti sırasıyla ConvLayer , PrimaryCaps , DigitCaps ve Decoder Fonksiyonlarına tabi tutulur. Bu sayede her veri DCnet algoritmasına göre eğitime hazır hale getirilmiş olur.

Capsnet Algoritmasının bir başka gerekliliği de forward (geri besleme) yapmasıdır. Bu sayede eğitim ile geri besleme yapılır ve her bir aşamadan çıkan sonuç ile bir sonraki aşamanın giriş bilgileri kıyaslanmış ve daha doğru bir sonuca erişilmiş olur.

Loss fonksiyonu eğitim sırasında eğitime girememiş veya belirli bir hatadan dolayı eğitim aşamalarında takılmış olan veri setlerini ayrıştırarak sonrasında eğitime tekrar sokmak için kullanılır. Daha sonra yapılacak işleme göre margin_loss veya reconstruction_loss fonksiyonlarına yönlendirme yapılır.

```
class Config:
    def _init_(self, dataset='mnist'):
        if dataset == 'mnist':
            # CNN (cnn)
            self.cnn_in_channels = 1
            self.cnn_out_channels = 256
            self.cnn_kernel_size = 9
           # Primary Capsule (pc)
            self.pc_num_capsules = 8
            self.pc_in_channels = 256
            self.pc_out_channels = 32
            self.pc_kernel_size = 9
            self.pc_num_routes = 32 * 6 * 6
           # Digit Capsule (dc)
            self.dc_num_capsules = 15
            self.dc_num_routes = 32 * 6 * 6
            self.dc_in_channels = 8
            self.dc_out_channels = 16
            # Decoder
            self.input_width = 28
            self.input_height = 28
elif dataset == 'your own dataset':
            pass
def train(model, optimizer, train_loader, epoch, amp_autocast, loss_scaler,
second_order):
    capsule_net = model
    capsule_net.train()
    targetTensor = torch.tensor(np.zeros((132, 15)))
    maskedTensor = torch.tensor(np.zeros((132, 15)))
    total_loss = 0
    for batch_id, (data, target) in enumerate(tqdm(train_loader)):
        data, target = Variable(data), Variable(target)
        target = target.view(target.shape[0], target.shape[2])
```

```
if USE_CUDA:
            data, target = data.cuda(), target.cuda()
        with amp_autocast():
            output, reconstructions, masked = capsule_net(data)
            loss = capsule_net.loss(data, output, target, reconstructions)
        optimizer.zero_grad()
        loss_scaler(loss, optimizer, clip_grad=None,
parameters=capsule_net.parameters(), create_graph=second_order)
        targetTensor[batch_id * BATCH_SIZE:(batch_id + 1) * BATCH_SIZE, :] =
target
        maskedTensor[batch_id * BATCH_SIZE:(batch_id + 1) * BATCH_SIZE, :] =
masked
        correctExamplesNum = sum(np.argmax(maskedTensor.data.cpu().numpy(), 1)
== np.argmax(targetTensor.data.cpu().numpy(), 1))
        total_loss += loss.item()
    print("Train accuracy:", correctExamplesNum / 132)
    print("Train Loss:", total_loss)
def test(capsule_net, test_loader, epoch, amp_autocast):
    capsule_net.eval()
    test_loss = 0
    targetTensorForTest = torch.tensor(np.zeros((33, 15)))
    maskedTensorForTest = torch.tensor(np.zeros((33, 15)))
    for batch_id, (data, target) in enumerate(test_loader):
```

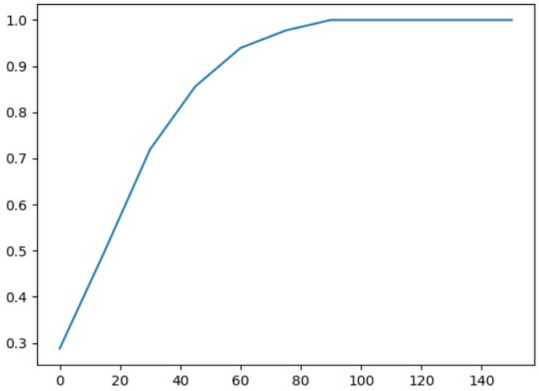
```
  data, target = Variable(data), Variable(target)
        target = target.view(target.shape[0], target.shape[2])
        if USE_CUDA:
            data, target = data.cuda(), target.cuda()
        with amp_autocast():
            output, reconstructions, masked = capsule_net(data)
            loss = capsule_net.loss(data, output, target, reconstructions)
        targetTensorForTest[batch_id * batchSizeTest:(batch_id + 1) *
batchSizeTest, :] = target
        maskedTensorForTest[batch_id * batchSizeTest:(batch_id + 1) *
batchSizeTest, :] = masked
        test_loss += loss.item()
    correctExamplesNumTest =
sum(np.argmax(maskedTensorForTest.data.cpu().numpy(), 1) ==

np.argmax(targetTensorForTest.data.cpu().numpy(), 1))
    return correctExamplesNumTest / 33
```

```
correctExamplesNumTest = sum(np.argmax(maskedTensorForTest.data.cpu().numpy(),
1) ==   np.argmax(targetTensorForTest.data.cpu().numpy(), 1))
    return correctExamplesNumTest / 33
from timm.utils import NativeScaler
from sklearn.model_selection import StratifiedKFold
if _name_ == '_main_':
    torch.manual_seed(1)
    dataset = 'mnist'
    finalAcc = np.zeros(5)
    fileLinks, labels = build_dataset()
    skf = StratifiedKFold(n_splits=5)
    for cv, (train_index, test_index) in enumerate(skf.split(fileLinks,
labels)):
        yale = Yale(BATCH_SIZE, batchSizeTest, train_index, test_index)
        config = Config(dataset)
        capsule_net = CapsNet(config)
        capsule_net = torch.nn.DataParallel(capsule_net)
```

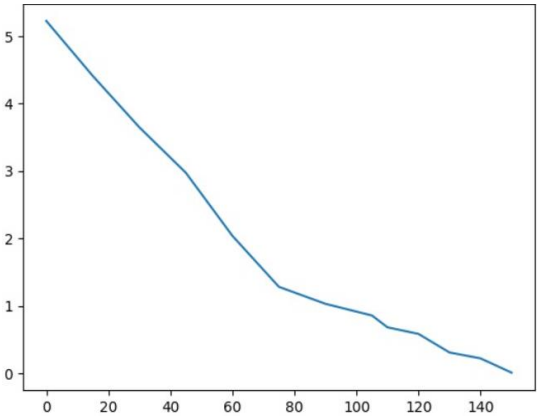
```
amp_autocast = torch.cuda.amp.autocast
        loss_scaler = NativeScaler()
        optimizer = torch.optim.Adam(capsule_net.parameters(), lr=0.001)
        second_order = hasattr(optimizer, 'is_second_order') and
optimizer.is_second_order
        if USE_CUDA:
            capsule_net = capsule_net.cuda()
        capsule_net = capsule_net.module
        for e in range(1, N_EPOCHS + 1):
            train(capsule_net, optimizer, yale.train_loader, e, amp_autocast,
loss_scaler, second_order)
            acc = test(capsule_net, yale.test_loader, e, amp_autocast)
            print("Fold = ", str(cv), "Epoch = ", str(e), "Test accuracy:",
acc)
            if (acc > finalAcc[cv]):
                finalAcc[cv] = acc
        print("Fold Acc = ", finalAcc[cv])
        print(finalAcc)
        print("Final Folds Mean Acc = ", np.mean(finalAcc))
```

Elde ettiğimiz sonuçlar:

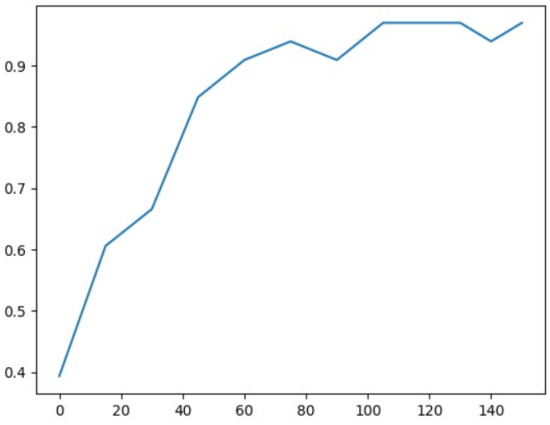
Fold[0] Train Accuracy 150 Epoch



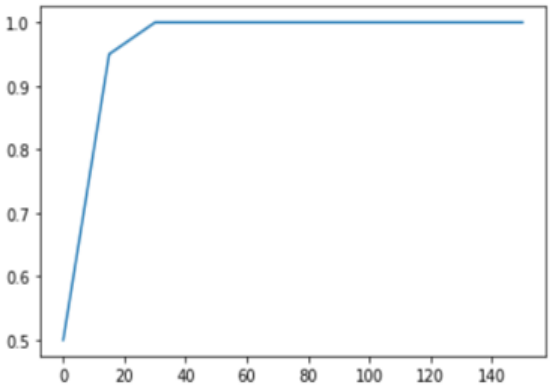
Fold[0] Train Loss 150 Epoch;



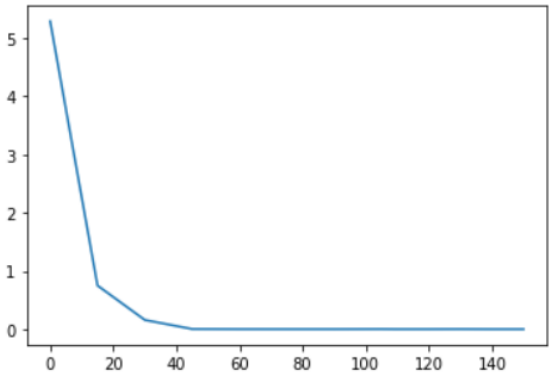
Fold[0] Test accuracy 150 Epoch;



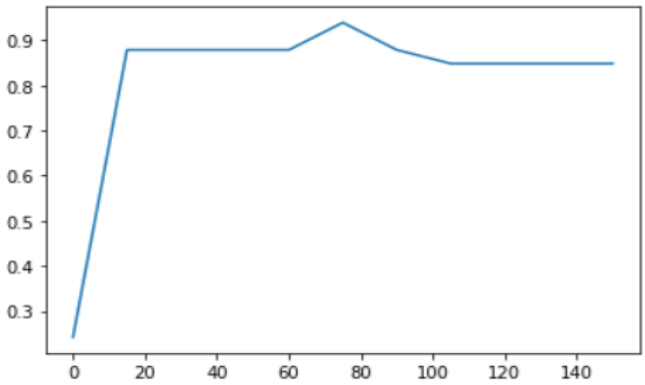
Fold[1] Train accuracy 150 Epoch



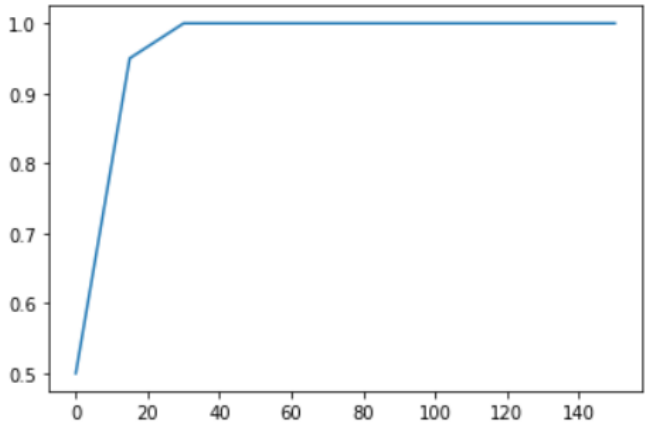
Fold[1] Train Loss 150 Epoch



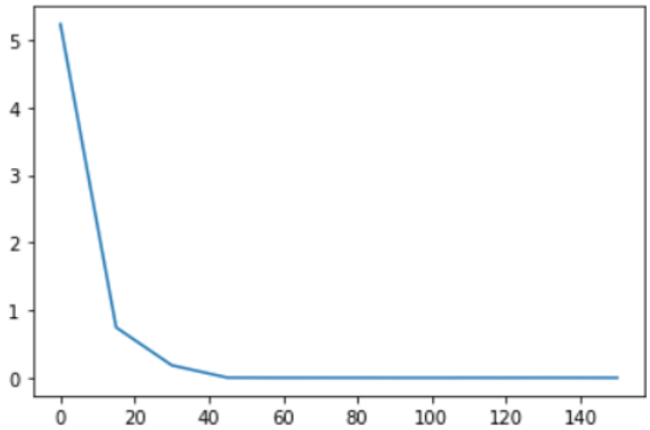
Fold[1] Test accuracy 150 Epoch;



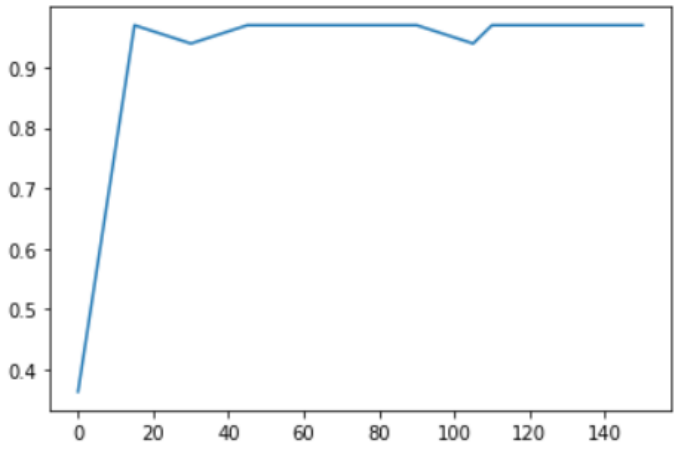
Fold[2] Train accuracy 150 Epoch;



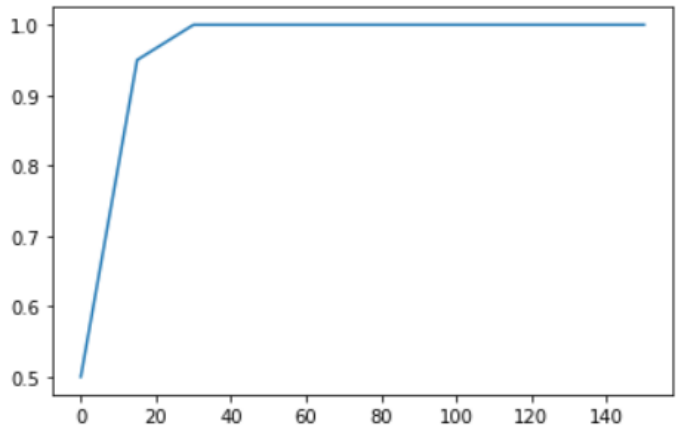
Fold[2] Train Loss 150 Epoch;



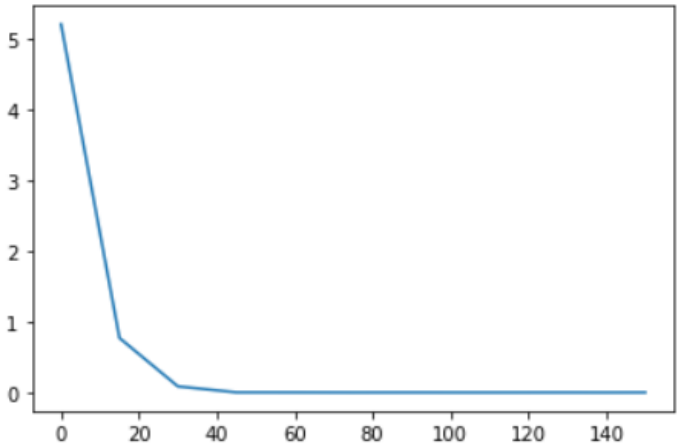
Fold[2] Test accuracy 150 Epoch;



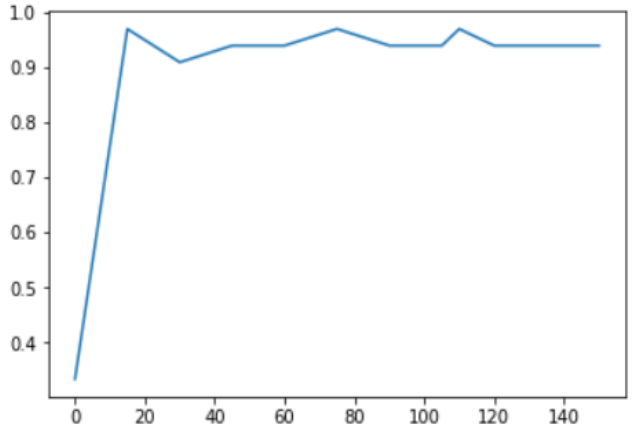
Fold[3] Train accuracy 150 Epoch;



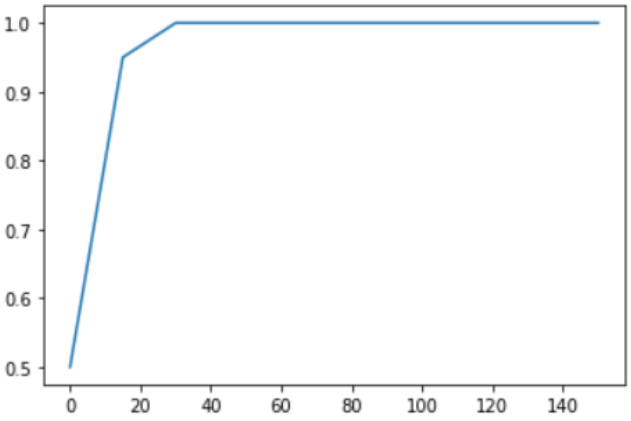
Fold[3] Train Loss 150 Epoch;



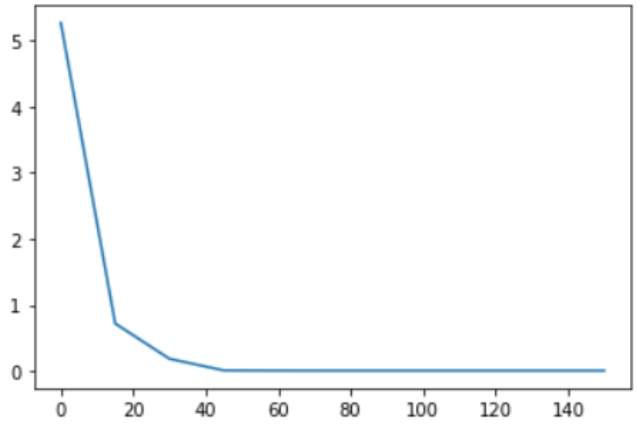
Fold[3] Test accuracy 150 Epoch;



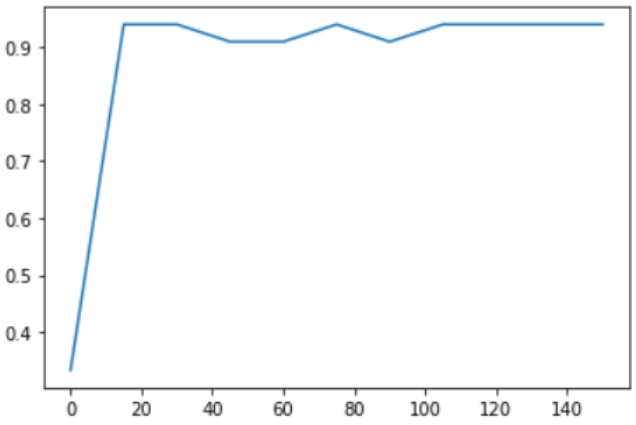
Fold[4] Train accuracy 150 Epoch;

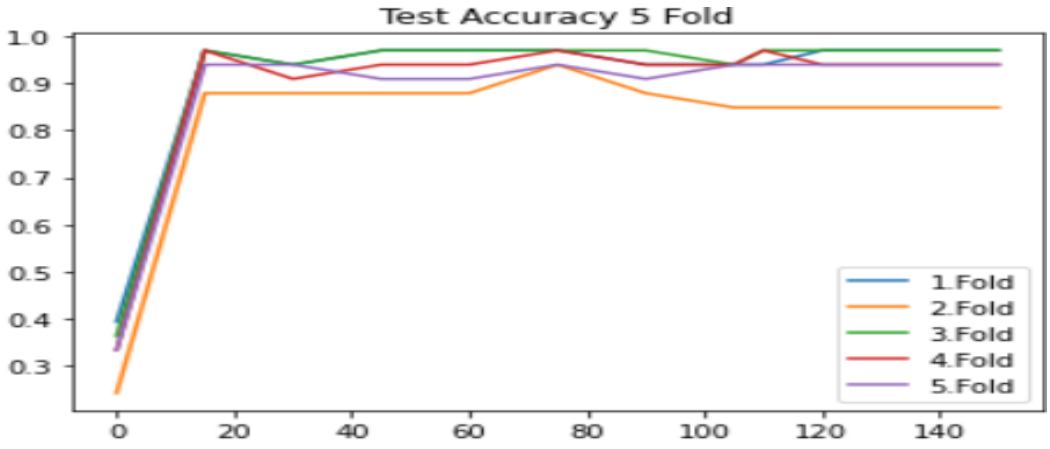


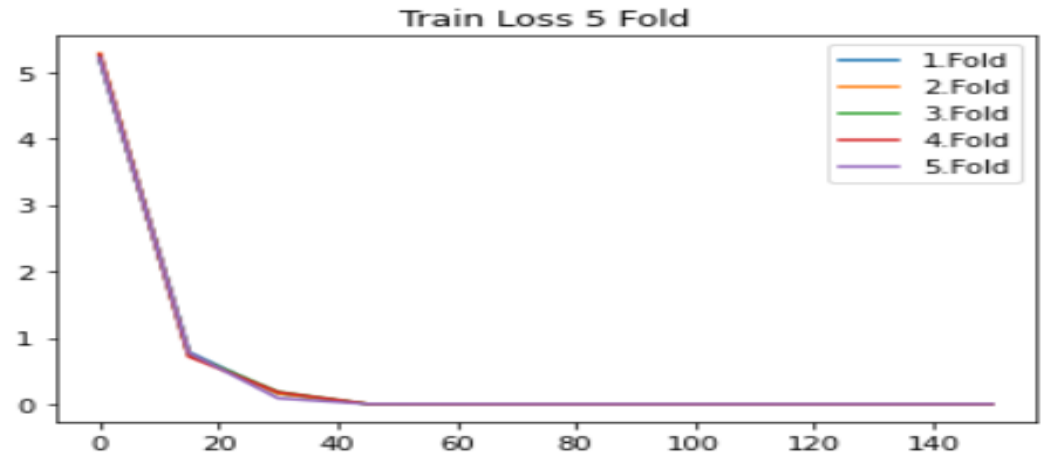
Fold[4] Train Loss 150 Epoch;



Fold[4] Test accuracy 150 Epoch;


0.9
0.8
0.7
0.6
0.5
0.4
0
20
40
60
80
100
120
140


Test Accuracy 5 Fold
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0
20
40
60
80
100
120
140
1.Fold
2.Fold
3.Fold
4.Fold
5.Fold


Train Loss 5 Fold
5
4
3
2
1
0
0
20
40
60
80
100
120
140
1.Fold
2.Fold
3.Fold
4.Fold
5.Fold

3.SONUÇLAR

Yale veri setine uyguladığımız her iki yöntemimimiz sonucunda;

Capsule Network üzerinde,

```
[0.93939394 0.96969697 0.93939394 0.90909091 0.93939394]

Acc =  0.9393939393939394
```

5 foldun ortalamasından doğruluk oranını **%93** olarak bulmuş bulmaktayız.

Efficent Capsnet Network üzerinde,

[0.21212121 0.18181818 0.21212121 0.18181818 0.15151515]
Acc = 0.18787878787878787

5 foldun ortalamasından doğruluk oranını **%18** olarak bulmuş bulmaktayız.

Dense Capsule Network üzerinde,

[1.0, 0.9090909090909091, 1.0, 0.9696969696969697 , 0.9696969696969697]

Acc = 0.9696969696969697

5 foldun ortalamasından doğruluk oranını **%96** olarak bulmuş bulmaktayız.

Sonuçları detaylı bir şekilde aşağıdaki linkler üzerinden inceleyebilirsiniz.

https://colab.research.google.com/drive/1FzkPaYascaNXAQm9mKAYJ2kZVSLyL9FZ?usp=sharing
https://colab.research.google.com/drive/1CRupxmFJInUUSpNDRSQHYiHxwwS0kP4f?usp=sharing

https://colab.research.google.com/drive/1o2_N-BvqoeksoK8IyBwW83QfH6KL0k0L?usp=sharing#scrollTo=iXA0k734cSdQ

4. KAYNAKLAR

1. Doç.Dr. Haldun AKPINAR, Yapay Sinir Ağları Gelişimi ve Yapılarının İncelenmesi, 1994.
2.Gao Huang, Zhuang Liu, Kilian Q Weinberger, and Laurens van der Maaten. Densely connected convolutional networks. In Proceedings of the IEEE conference on computer vision and pattern recognition, volume 1, page 3, 2017.
3. Guangcong Sun,Shifei Ding,Tongfeng Sun,Chenglong Zhang,Wei Du,2021 URL https://link.springer.com/article/10.1007/s10489-021-02630-w
4.Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 770–778, 2016.
5. Mingxing Tan, Quoc V. Le, EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks.
6.Parnian Afshar, Arash Mohammadi, and Konstantinos N Plataniotis. Brain tumor type classification via capsule networks. arXiv preprint arXiv:1802.10200, 2018.
7.Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. CoRR, abs/1505.00387, 2015. URL http://arxiv.org/abs/1505.00387.
8.Sai Samarth R. Phaye, Apoorva Sikka, Abhinav Dhall, Deepti Bathula/1805.04001v1,2018.URL https://arxiv.org/pdf/1805.04001v1.pdf
9. Sara Sabour, Nicholas Frosst, Geoffrey E. Hinton, Dynamic Routing Between Capsules,2017.
10. Vasıf NABİYEV, Yapay Zekâ, 2021, 607-615.
11. Vittorio Mazzia, Francesco Salvetti, Marcello Chiaberge, EFFICIENT-CAPSNET: Capsule Network With Self-Attention Routing.

STANDARTLAR ve KISITLAR FORMU

Projenin hazırlanmasında uyulan standart ve kısıtlarla ilgili olarak, aşağıdaki soruları cevaplayınız.

- Projenizin tasarım boyutu nedir? (Yeni bir proje midir? Var olan bir projenin tekrarı mıdır? Bir projenin parçası mıdır? Sizin tasarımınız proje toplamının yüzde olarak ne kadarını oluşturmaktadır?)

Yüz tanıma ve birçok farklı şeylerin tanınması için kullanılan derin öğrenme yöntemlerinden biri olan kapsül ağlarının kullanımı üzerine yapılan bir projedir. Yeni bir proje değildir ve bir projenin devamı değildir. Öncesinde kodlanmış yöntemleri kendimiz kodlamaya ve yale veri seti üzerinden başarı sağlamayı amaçlamış bulunmaktayız.

- Projenizde bir mühendislik problemini kendiniz formüle edip, çözdünüz mü? Açıklayınız.

Var olan matematiksel yöntemleri anlayıp kodlamasını gerçekleştirdik.

- Önceki derslerde edindiğiniz hangi bilgi ve becerileri kullandınız?

Bilgisayar Mühendisliğine giriş dersi, Algoritmalar, Veri Madenciliği, Yapay Zekâ ve Yazılım Mühendisliğine giriş derslerinde edindiğimiz programlama bilgilerini, mantığını, algoritma geliştirme becerilerini, veri ön işleme, sınıflandırma ve dökümantasyon oluşturma becerilerimizi kullandık.

- Kullandığınız veya dikkate aldığınız mühendislik standartları nelerdir? (Proje konunuzla ilgili olarak kullandığınız ve kullanılması gereken standartları burada kod ve isimleri ile sıralayınız).

IEEE/IE 12207 Standardı ile belirtilen yazılım yaşam döngüsü süreçleri dikkate alınmıştır.

- Kullandığınız veya dikkate aldığınız gerçekçi kısıtlar nelerdir? Lütfen boşlukları uygun yanıtlarla doldurunuz. (Ekonomi, Çevre sorunları, Sürdürülebilirlik, Üretilebilirlik, Etik, Sağlık, Güvenlik, Sosyal ve politik sorunlar)

Proje geliştirme süreçlerinde kullanılan pycharm Spyder gibi geliştirme ortamının community versiyonu kullanıldığı için bir ücretlendirme yapılmamıştır. Kaggle üzerinden ücretsiz olan paylaşılan Starter: Yale Face Database c5f3978b-5 kullanılmıştır ve herhangi

bir ücret ödenmemiştir. Hızlı test için de Google Collab kullanılmıştır. Projemiz çevre için herhangi bir sorun teşkil etmemektedir.
Projeyi başarılı bir şekilde tamamlayabilirsek kullanıcı feedbackleri ile sürdürülebililirliği sağlamayı amaçlıyoruz.
Projenin sonunda elde ettiğimiz başarı oranına göre üretkenliği amaçlamayı hedeflemekteyiz.
İnsani değere zarar verecek etik açıdan herhangi bir içeriğe sahip değildir.
Projemiz sağlık açısından herhangi bir sorun teşkil etmemektedir.
Projede kullanıcının herhangi bir verisine izinsiz erişim olmadığından dolayı güvenlik için bir sorun teşkil etmeyecektir.
Projemizin sosyal açıdan sorununu henüz tespit edememekle beraber politik açıdan bir sorun teşkil etmemektedir.