
buzdağı

Yayın Numarası: 2101

1. Baskı: Ağustos 2021

ISBN 978-605-06248-2-3

Sertifika Numarası: 41825
Buzdağı Yayınevi
Altay Mahallesi 2668. Cadde
Fırat Life Style Rezidans 1F/61
Eryaman/ANKARA
t: +90 312 219 77 98
f: +90 312 219 55 43
info@buzdagiyayinevi.com

Baskı
Ankamat Matbaacılık
46700

Scikit-Learn, Keras ve TensorFlow ile Uygulamalı Makine Öğrenmesi

Aurélien Géron

Genel Yayın Yönetmeni
Fatih ÖZDEMİR

Yazar
Aurélien Géron

Çevirmen
Bilgin AKSOY
Özgür KAYA

Editör
Arş. Gör. Dr. Mustafa Murat ARAT
Doç. Dr. Vedat ÇELİK

Son Okuma
Muhammed TOPRAK
Ahmet Okan ŞEKER

Dizgi
Veysel TOPRAK

İçindekiler

1 Makine Öğrenmesi Dünyası

İnsanlar "Makine Öğrenmesi"ni duyduklarında akıllarına bir robot geliyor: Kime sorduğunuza bağlı olarak değişen, güvenilir bir uşak ya da ölümcül bir Terminatör. Ama makine öğrenmesi geleceğe ait bir hayalden ibaret değil: Hâlihazırda elimizde bulunuyor. Aslında on yıllardan beri Optik Karakter Tanıma (OCR) gibi bazı özel uygulamalarıyla hayatımızdadır. Ana akım hâline gelen ilk makine öğrenmesi uygulaması 1990'ların başından beri yüz milyonlarca insanın hayatını iyileştiren *gereksiz posta filtresidir*. Tam olarak kendini tanıyan bir Skynet değil ancak teknik olarak makine öğrenmesi olarak nitelendirilmektedir (aslında bir e-postayı artık gereksiz posta olarak işaretlemeniz gerektiğini çok da iyi öğrendi). Bunu daha iyi önerilerden, sesli aramaya kadar düzenli olarak kullandığınız yüzlerce ürün ve özelliğe sessizce güç veren yüzlerce makine öğrenmesi uygulaması izledi.

Makine öğrenmesi nerede başlar ve nerede biter? Bir makinenin bir şeyi *öğrenmesi* tam olarak ne anlama gelmektedir? Wikipedia'nın bir kopyasını indirirsem bilgisayarım gerçekten bir şey öğrenecek mi? Aniden daha akıllı mı oldu? Bu bölüme makine öğrenmesinin ne olduğunu ve neden kullanmak isteyebileceğinizi açıklayarak başlayacağız.

Ardından, makine öğrenmesi kıtasını keşfetmeye başlamadan önce haritaya bir göz atacağız ve ana bölgeler ve en önemli yerler hakkında bilgi edineceğiz: Denetimli ve denetimsiz öğrenmeyi, çevrim içi ve yığın öğrenmeyi, örnek tabanlı ve model tabanlı öğrenmeyi göreceğiz. Ardından, tipik bir makine öğrenmesi projesinin iş akışına bakacağız, karşılaşabileceğiniz ana zorlukları tartışacağız ve bir Makine Öğrenim sisteminin nasıl değerlendirileceğini ve hassas ayarın nasıl yapılacağını ele alacağız.

Bu bölüm, her veri bilimcinin ezbere bilmesi gereken birçok temel kavramı (ve jargonu) içermektedir. Bu, oldukça basit bir tanıtım olacak ancak kitabın geri kalanına devam etmeden önce her şeyin sizin için net olduğundan emin olmalısınız. Öyleyse bir kahve alın ve başlayalım!

Eğer makine öğrenmesi temellerini zaten biliyorsanız doğrudan Bölüm 2'ye atlayabilirsiniz. Ama emin değilseniz devam etmeden önce bölümün sonunda listelenen tüm soruları yanıtlamaya çalışın.

1.1 Makine Öğrenmesi Nedir?

Makine öğrenmesi, verilerden öğrenen bilgisayarları programlama bilimidir (ve sanatı).

İşte biraz daha genel bir tanım:

> [Makine Öğrenmesi], bilgisayarlara açıkça programlama yapmadan öğrenme yeteneği kazandıran bir çalışma alanıdır.
>
> — *Arthur Samuel, 1959*

Ve daha mühendislik odaklı olanıysa:

> Belli bir T görevi ve P performans ölçütü için eğer P performans ölçütü, T görevinde E tecrübeleriyle artıyorsa bu E tecrübelerinden öğrendiği söylenen bilgisayar programıdır.
>
> — *Tom Mitchell, 1997*

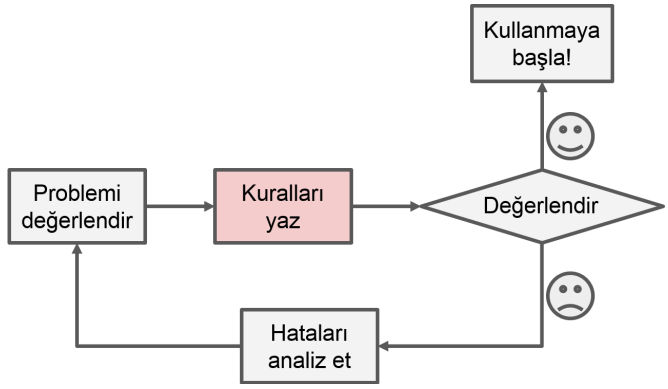
Gereksiz posta filtreniz, gereksiz posta (örneğin kullanıcılar tarafından işaretlenmiş) örnekleri ve normal ("gereksiz olmayan") posta örnekleri verilip gereksiz posta işaretlemeyi öğrenebilen bir makine öğrenmesi programıdır. Sistemin öğrenmek için kullandığı örneklere *eğitim veri seti* denir. Her bir eğitim örneğine *örneklem* adı da verilir. Bu durumda T görevi yeni postaları gerekli/gereksiz olarak işaretlemek, E tecrübesi *eğitim veri seti* ve P performans ölçütüyse, örneğin doğru sınıflandırılan posta oranı olarak tanımlanabilir. Bu performans ölçütü *doğruluk oranı* olarak adlandırılır ve genellikle sınıflandırma görevlerinde kullanılır.

Wikipedia'nın bir kopyasını indirirseniz artık bilgisayarınızda çok daha fazla veri bulunacaktır. Ama yaptığı herhangi bir görevde daha iyi değildir. Bu nedenle, Wikipedia'nın bir kopyasını indirmek makine öğrenmesi değildir.

1.2 Neden Makine Öğrenmesi Kullanılmalı?

Geleneksel programlama tekniklerini kullanarak nasıl gereksiz posta filtresi yazacağınızı düşünün (Şekil-1.1):

1. Öncelikle gereksiz postanın tipik olarak nasıl göründüğünü düşünürdünüz. Bazı kelimelerin veya kelime öbeklerinin ("4U", "kredi kartı", "ücretsiz" ve "şaşırtıcı" gibi) konu satırında çok fazla ortaya çıkma eğiliminde olduğunu fark edebilirsiniz. Belki de gönderenin adında, postanın gövdesinde ve/veya postanın diğer bölümlerinde başka örüntüler de fark edebilirsiniz.
2. Fark ettiğiniz her bir örüntü için bir tespit algoritması yazarsınız ve bu örüntülerden birkaçı tespit edilirse programınız postaları gereksiz olarak işaretler.
3. Programınızı test eder ve üretim ortamında kullanmaya başlamadan önce yeterince iyi olana kadar 1. ve 2. adımları tekrarlarsınız.



Şekil 1.1: *Geleneksel Yaklaşım*

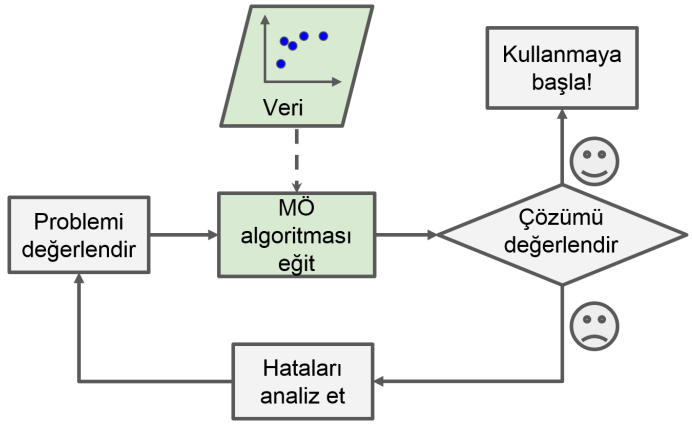
Problem zor olduğundan programınız muhtemelen karmaşık kuralların uzun bir listesi hâline gelecektir ve bakımı oldukça zorlaşacaktır.

Aksine, makine öğrenmesi tekniklerine dayanan bir gereksiz posta filtresi, gerekli posta örneklerine kıyasla gereksiz posta örneklerinde olağan dışı sık kullanılan kelime örüntülerini tespit ederek hangi kelimelerin ve ifadelerin gereksiz posta için iyi tahminci olduğunu otomatik olarak öğrenir (Şekil-1.2). Program çok daha kısa, bakımı daha kolay ve büyük olasılıkla daha doğru sonuçlar getirecektir.

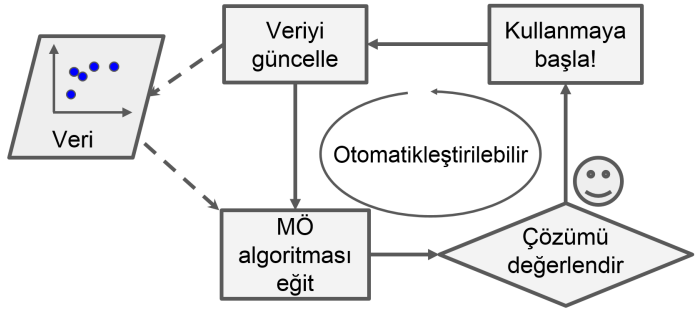
Gereksiz posta gönderenler "4U" içeren tüm postalarının engellendiğini fark ederse ne olur? Bunun yerine "For U" yazmaya başlayabilirler. Geleneksel

programlama tekniklerini kullanan bir gereksiz posta filtresinin "For U" içeren postaları işaretlemek üzere güncellenmesi gerekir. Gereksiz posta gönderenler gereksiz posta filtreniz üzerinde çalışmaya devam ederse sonsuza kadar yeni kurallar yazmaya devam etmeniz gerekir.

Buna karşılık makine öğrenmesi tekniklerine dayanan bir gereksiz posta filtresi kullanıcılar için işaretlenen gereksiz postada "For U" nun olağan dışı bir şekilde sık sık geldiğini otomatik olarak fark eder ve sizin müdahaleniz olmadan bunları gereksiz posta olarak işaretlemeye başlar (Şekil-1.3).



Şekil 1.2: *Makine öğrenmesi yaklaşımı*

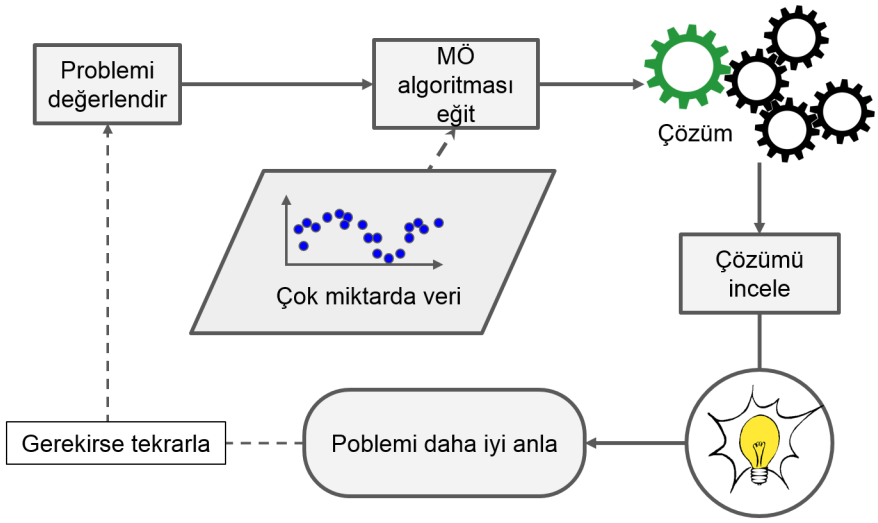


Şekil 1.3: *Otomatik olarak değişime uyum sağlaması*

Makine öğrenmesinin yıldızlaştığı bir başka alan geleneksel yaklaşımlar için çok karmaşık olan ya da bilinen bir çözüm algoritması bulunmayan problemlerdir. Örneğin, bir konuşma tanıma programı düşünün. Basit bir başlangıç yapmak ve "bir" ve "iki" kelimelerini ayırt edebilecek bir program yazmak istediğinizi varsayalım. "İki" kelimesinin yüksek perdeli bir sesle ("İ") başladığını fark edebilirsiniz, böylece yüksek perdeli ses yoğunluğunu ölçen bir algoritmayı kodlayabilir, birleri ve ikileri ayırt etmek için kullanabilirsiniz. Ama açıkçası

bu teknik gürültülü ortamlarda ve düzinelerce dilde milyonlarca farklı insanın konuştuğu binlerce kelimeye ölçeklenemez. En iyi çözüm (en azından bugün) her kelime için birçok örnek kayıt verildiğinde kendi kendine öğrenen bir algoritma yazmaktır.

Son olarak, makine öğrenmesi insanların öğrenmesine yardımcı olabilir (Şekil-1.4). Makine öğrenmesi algoritmaları ne öğrendiklerini görmek için incelenebilir (gerçi bu bazı algoritmalar için biraz zor olabilir). Örneğin, bir gereksiz posta filtresi yeteri kadar gereksiz posta üzerinde eğitildikten sonra gereksiz postaları en iyi tahmin ettiğine inandığı kelime listesi ve kelime kombinasyonları incelenebilir. Bazen bu; şüphe edilmeyen korelasyonlar ya da yeni eğilimler ortaya çıkartır. Böylece problemin daha iyi anlaşılmasına yol açar. Makine öğrenmesi tekniklerinin büyük miktarda veriye uygulanması kolayca görülmeyen örüntülerin keşfedilmesine yardımcı olabilir. Buna *veri madenciliği* denir.



Şekil 1.4: *Makine öğrenmesi, insanların öğrenmesine yardımcı olabilir*

Özetlemek gerekirse makine öğrenmesi aşağıda sıralanan konular için harika bir seçimdir:

- Mevcut çözümlerin çok sayıda hassas ayar veya uzun kural listesi gerektirdiği sorunlar: Bir makine öğrenmesi algoritması genellikle kodu basitleştirebilir ve geleneksel yaklaşımdan daha iyi performans gösterebilir.

- Geleneksel bir yaklaşım kullanmanın iyi bir çözüm getirmediği karmaşık sorunlar: En iyi makine öğrenmesi teknikleri belki bir çözüm bulabilir.

2.2.2 Performans Ölçütü Seçin

Sonraki adımınız bir performans ölçütü seçmektir. Bağlanım problemleri için tipik bir performans ölçütü Ortalama Karesel Hata Karekökü'dür (RMSE). Sistem tahminlerini yaptığında büyük hatalara daha büyük ağırlıklar vererek ne kadar hata yaptığı hakkında bir fikir verir. Denklem-2.1 matematiksel olarak RMSE'nin nasıl hesaplanacağını göstermektedir.

Ortalama Karesel Hata Karekökü (RMSE)

$$\text{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right)^2} \tag{2.1}$$

Notasyon
Bu denklem bu kitap boyunca kullanacağımız birçok makine öğrenmesi notasyonunu içermektedir.

- m, üzerinde RMSE hesabını yapacağınız veri setindeki örnek sayısıdır.
  - Örneğin eğer doğrulama veri setinde 2000 bölge için RMSE hesaplıyorsanız $m = 2000$'dir.
- $\mathbf{x}^{(i)}$ veri setindeki i^{inci} örneğin tüm niteliklerinin (etiketi hariç) değerlerini içeren bir vektördür ve $y^{(i)}$ ise o örneğin etiketidir (o örnek için istenen çıktı değeri).
  - Örneğin ilk bölge -118.29°enleminde, 33.91°boylamında ve gelirlerinin medyanı 38372$ olan 1416 nüfusa ve medyan konut fiyatı 156400$ (şimdilik diğer nitelikleri atlıyoruz) olsun:

$$\mathbf{x}^{(1)} = \begin{pmatrix} -118.29 \\ 33.91 \\ 1416 \\ 38372 \end{pmatrix}$$

ve:

$$y^{(1)} = 156400$$

- $\mathbf{X}$ veri setindeki tüm örneklerin tüm niteliklerini (etiketleri hariç) içeren bir matristir. Her satır bir örneğe ait ve i^{inci} satır $\mathbf{x}^{(i)}$

vektörünün transpozuna, yani $(\mathbf{x}^{(i)})^\top$ ifadesine eşittir.[4]

– Örneğin ilk bölgenin az önce tanımladığımız gibi olduğunu düşünürsek $\mathbf{X}$ matrisi şuna benzer:

$$\mathbf{X} = \begin{pmatrix} (\mathbf{x}^{(1)})^\intercal \\ (\mathbf{x}^{(2)})^\intercal \\ \vdots \\ (\mathbf{x}^{(1999)})^\intercal \\ (\mathbf{x}^{(2000)})^\intercal \end{pmatrix} = \begin{pmatrix} -118.29 & 33.91 & 1416 & 38372 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

- h, hipotez de denilen sisteminizin tahmin fonksiyonudur. Sisteminize bir örneğe ait nitelik vektörü $\mathbf{x}^{(i)}$ verildiğinde, çıktısı o örneğe ait tahmin $\hat{y}^{(i)} = h(\mathbf{x}^{(i)})$ edilen değerdir $\hat{y}$.
  – Örneğin eğer sisteminiz ilk bölge için medyan konut fiyatını 158400$ olarak tahmin ederse $\hat{y}^{(1)} = h(\mathbf{x}^{(1)}) = 158400\$$'dır. O bölge için tahmin hatası ise $\hat{y}^{(1)} - y^{(1)} = 2000$'dir.
- RMSE($\mathbf{X}$,h) h hipotezinin bir örnekler seti için ölçülen maliyet fonksiyonudur.

Skaler değerleri (m ya da $y^{(i)}$ gibi) ve fonksiyonları isimlendirirken (h gibi) küçük yatık harfler, vektörleri ($\mathbf{x}^{(i)}$ gibi) küçük kalın harfler ve matrisleri ($\mathbf{X}$ gibi) büyük kalın harflerle gösteriyoruz.

Her ne kadar RMSE bağlanım görevlerinde genellikle tercih edilen bir performans ölçütü olsa da bazen farklı fonksiyonlar da tercih edebilirsiniz. Örneğin çok fazla bölge için aykırı değerlerin olduğunu düşünün. Bu durumda *ortalama mutlak hata* kullanmayı düşünebilirsiniz (MAE, ortalama mutlak değişim olarak da adlandırılır; Denklem-2.2 bakınız):

Ortalama mutlak hata (MAE)

$$\text{MAE}(\mathbf{X}, h) = \frac{1}{m} \sum_{i=1}^{m} \left| h\left(\mathbf{x}^{(i)}\right) - y^{(i)} \right| \tag{2.2}$$

Hem RMSE hem de MAE iki vektör arasındaki mesafeyi ölçmenin bir yoludur: Tahmin vektörü ve hedef etiket vektörü. Farklı mesafe ölçütleri ya da *normları* kullanmak mümkündür:

- Karelerin toplamının karekökünü (RMSE) hesaplamak şu anda aşina olduğunuz *Öklid normu* (ÇN: Euclidean norm) hesaplamaya karşılık gelir.

[4] Transpoz operatörü sütun vektörünü satır vektörüne dönüştürür (ve tam tersine).

Aynı zamanda ℓ_2 *norm* olarak adlandırılır ve $\|\cdot\|_2$ ile de gösterilir. (Ya da sadece $\|\cdot\|$).

- Mutlak değerlerin toplamını (MAE) hesaplamak ℓ_1 normuna karşılık gelir ve $\|\cdot\|_1$ ile gösterilir. Bazen Manhattan normu olarak da adlandırılır. Çünkü bir şehirdeki iki nokta arasındaki mesafeyi sadece ortogonal bloklar kullanarak ölçer.
- Daha genel olarak n tane eleman içeren $\mathbf{v}$'nin ℓ_k normu vektörü $\|\mathbf{v}\|_k = \left(|v_0|^k + |v_1|^k + \ldots + |v_n|^k\right)^{1/k}$ olarak tanımlanabilir. ℓ_0 vektörde sıfırdan farklı elemanları gösterirken, ℓ_∞ en büyük mutlak değeri gösterir.
- Büyük norm indeksleri küçük değerleri yok sayarken büyük değerlere odaklanır. Bu yüzden, RMSE aykırı değerlerden MAE'ye göre daha fazla etkilenir. Ama aykırı değerler üssel olarak seyrek olduğunda (çan eğrisinde olduğu gibi) RMSE çok başarılı olur ve tercih edilir.

2.2.3 Varsayımları Kontrol Edin

Son olarak; şu ana kadar yapılan varsayımları (sizin tarafınızdan ya da başkaları tarafından) listelemek ve doğrulamak iyi olacaktır ki bu size yapacağınız ciddi hataları önceden görmenizde yardımcı olacaktır. Örneğin, sisteminizin çıktısı olacak ve başka bir makine öğrenmesi sistemine girdi olacak olan bölge fiyatlarının sizin ürettiğiniz şekilde kullanılacağını varsaydınız. Ama diğer sistem ürettiğiniz fiyatları kullanmak yerine kategorilere (mesela "ucuz", "ortalama" ya da "pahalı") dönüştürüp kullanacaksa ne olacak? Bu durumda fiyatları mükemmel bir şekilde tahmin etmek çok da önemli olmayacak onun yerine sadece doğru kategoriyi tahmin etmesi gerekecek. Eğer öyleyse problem bağlanım görevi yerine sınıflandırma görevi olarak düşünülmeliydi. Bunu bir bağlanım görevi olarak düşünüp aylarca uğraştıktan sonra fark etmek istemezsiniz.

Neyse ki diğer sistemden sorumlu olan takımla yaptığınız görüşme sonucunda sadece kategorilere değil gerçek fiyatlara da ihtiyaçları olduğuna emin oldunuz. Çok güzel! Her şeyi hazırladınız, yeşil ışık yandı ve kodlamaya başlayabilirsiniz!

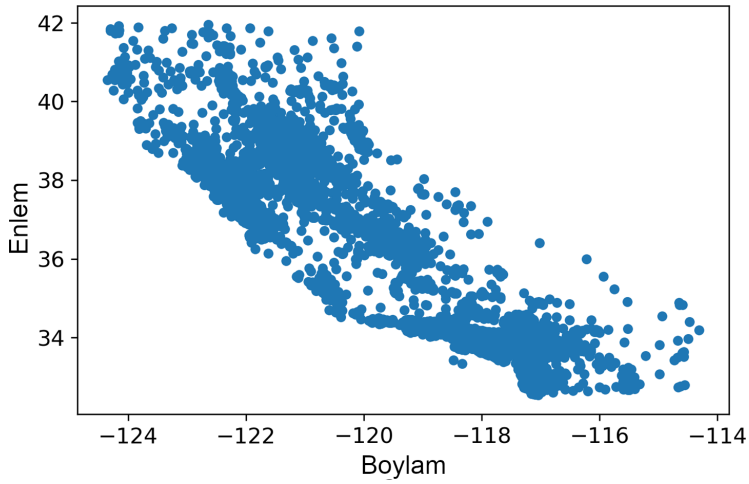
2.3 Verileri Toplayın

Ellerinizi kirletme zamanı geldi. Dizüstü bilgisayarınızı alıp Jupyter not defterinde bulunan kod örneklerinin üzerinden geçmekten çekinmeyin. Tüm Jupyter not defteri örneklerini *https://github.com/ageron/handson-ml2* bağlantısında bulabilirsiniz.

Coğrafi Veriyi Görselleştirme

Enlem ve boylam gibi coğrafi bilgiler bulunduğundan bölge verilerini dağılım grafiği kullanarak görselleştirmek iyi bir fikir olacaktır (Şekil-2.11'e bakınız):

```
housing.plot(kind="scatter", x="enlem", y="boylam")
```



Şekil 2.11: *Verinin coğrafi dağılım grafiği*

Evet grafik Kaliforniya'ya benziyor ama ondan başka bir örüntü görülemiyor. `alpha` opsiyonunu 0.1 yapmak verinin yoğun olduğu noktaları daha iyi görselleştirir (Şekil-2.12):

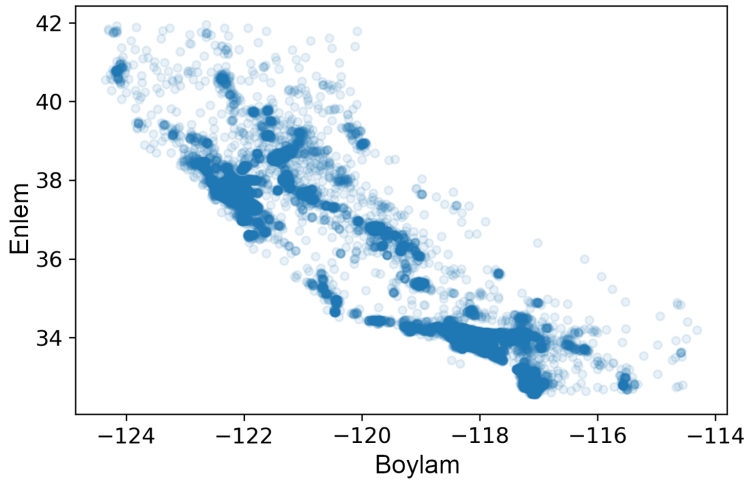
```
housing.plot(kind="scatter", x="enlem", y="boylam", alpha=0.1)
```

Şimdi daha iyi oldu: Yüksek yoğunluklu alanları görüyorsunuz. Los Angeles ve San Diego yakınlarında bulunan Bay Area, Sacramento ve Fresno yakınlarında uzanan Central Valley alanlarında yoğunluk göze çarpıyor.

Beynimiz resimlerdeki örüntüleri bulma konusunda çok iyidir ama görselleştirme parametreleriyle biraz daha oynayarak örüntüleri ortaya çıkarmanız gerekecektir.

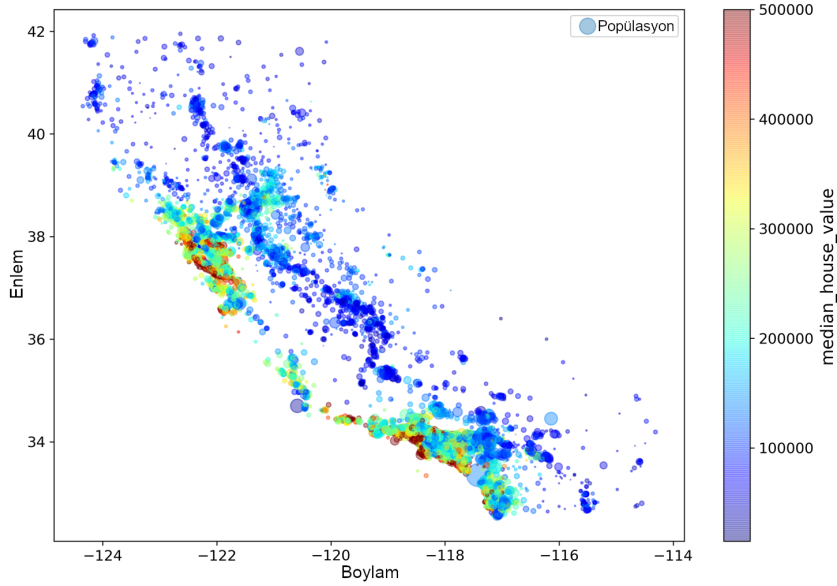
Şimdi de ev fiyatlarına bakalım (Şekil-2.13). Her dairenin yarıçapı nüfusu (`s` opsiyonu) renklerse fiyatları (`c` opsiyonu) gösteriyor. Maviden (düşük fiyatlar) kırmızıya (yüksek fiyatlar) değişen jet isimli, önceden tanımlı bir renk haritası (`cmap` opsiyonu) kullanacağız:[16]

[16] Bunu gri tonlamalı olarak okuyorsanız, kırmızı bir kalem alın ve Körfez Bölgesi'nden San Diego'ya kadar sahil şeridinin büyük bir kısmını karalayın (beklediğiniz gibi). Sacramento'nun etrafına da sarı bir yama ekleyebilirsiniz.



Şekil 2.12: *Yüksek yoğunluklu bölgeleri daha iyi görselleştirme*

```
housing.plot(kind="scatter", x="enlem", y="boylam", alpha=0.4,
    s=housing["population"]/100, label="nüfus", figsize=(10,7),
    c="median_house_value", cmap=plt.get_cmap("jet"), colorbar=True,
)
plt.legend()
```



Şekil 2.13: *Kaliforniya ev fiyatları: Kırmızı pahalı, mavi ucuz fiyatları, geniş daireler nüfus yoğun bölgeleri belirtmektedir.*

öznitelik sayısına göre doğrusaldır. Diğer bir deyişle iki katı örnek üzerinde (ya da iki katı öznitelik üzerinde) tahmin yapmak iki katı zaman alacaktır.

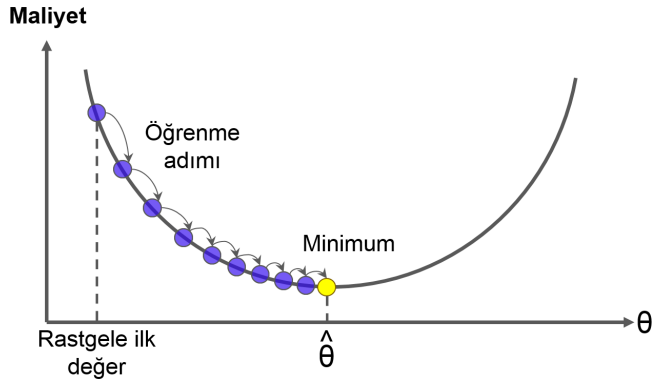
Şimdi doğrusal bağlanım modelini eğitmenin çok fazla sayıda öznitelik bulunduğunda ya da hafızaya sığmayacak kadar fazla örnek bulunduğunda çok kullanışlı olan oldukça farklı bir yöntemine bakacağız.

4.2 Gradyan İnişi

Gradyan inişi çok farklı problemler için optimum sonucu bulabilme yeteneğine sahip bir eniyileme algoritmasıdır. Gradyan inişinin ana fikri parametreleri her iterasyonda ayarlayarak maliyet fonksiyonunu enküçültmektir.

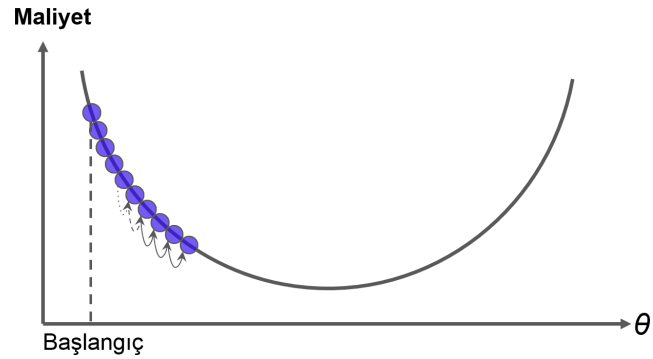
Bir dağda yoğun siste kaybolduğunuzu ve sadece ayağınızı bastığınız yerin eğimini hissettiğinizi düşünün. Vadiye hızla ulaşmanın iyi bir yolu en dik eğimi aşağı doğru takip etmektir. Bu tam olarak gradyan inişinin yaptığı şeydir: Hata fonksiyonunun θ parametre vektörüne göre gradyanını hesaplar ve azalan gradyan yönünde ilerler. Gradyan sıfır olduğunda, minimum seviyeye ulaştınız!

Somut olarak θ vektörünü rastgele değerlerle doldurarak (buna rastgele *ilk değer atama* denir) başlarsınız. Sonra yavaş yavaş onu geliştirirsiniz, her seferince küçük bir adım atıp, her adımda maliyet fonksiyonu (örneğin MSE) değerini azaltıp, bir minimuma yakınsayıncaya kadar ilerlersiniz (Şekil-4.3'e bakınız).



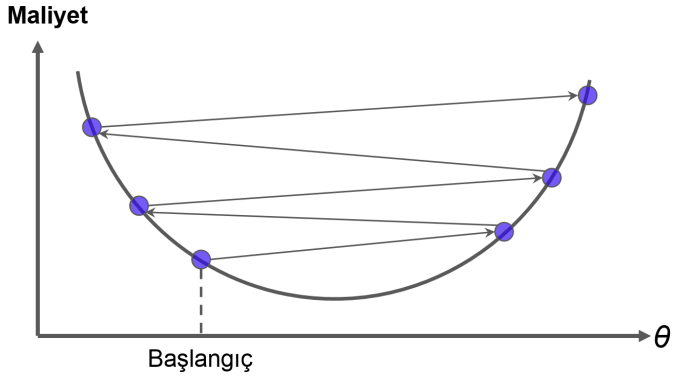
Şekil 4.3: *Bu gradyan inişi tasvirinde, model parametrelerinin rastgele olarak ilk ataması yapılır ve maliyet fonksiyonunu enküçültmek için defalarca ayarlanır; öğrenme adamının büyüklüğü, maliyet fonksiyonunun eğimiyle doğru orantılıdır. Dolayısıyla adımlar, parametreler minimuma yaklaştıkça yavaş yavaş küçülür.*

Gradyan inişinin önemli bir parametresi öğrenme oranı hiperparametresiyle kontrol edilen adım büyüklüğüdür. Eğer öğrenme oranı çok küçükse algoritma yakınsayana kadar çok fazla iterasyon yapacak ve bu çok zaman alacaktır (Şekil-4.4'e bakınız).



Şekil 4.4: *Öğrenme oranı çok küçük*

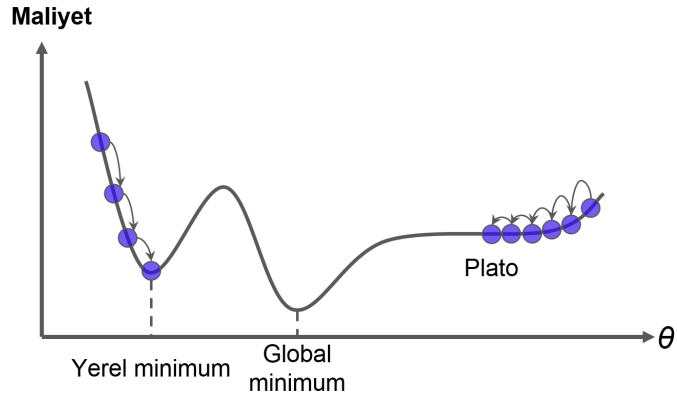
Diğer taraftan öğrenme oranı çok büyükse vadiden karşıya atlayıp, bulunduğunuz yerden daha yüksek bir noktaya ulaşabilirsiniz. Bu durum algoritmanın ıraksamasına sebep olur ve daha büyük değerlere doğru yönlendirir ve iyi bir çözüm bulamayabilir (Şekil-4.5'e bakınız).



Şekil 4.5: *Öğrenme oranı çok büyük*

Son olarak, tüm maliyet fonksiyonları düzgün biçimli bir çanak şekline sahip değildir. Delikler, sırtlar, platolar ve diğer düzensiz yüzeyler bulunabilir ve minimuma yakınsamayı zorlaştırabilir. Şekil-4.6 gradyan inişinin karşılaştığı iki ana zorluğu göstermektedir. Eğer rastgele ilk değer atama algoritmayı solda başlatırsa *yerel minimuma* ulaşır ki *global minimum* kadar iyi değildir. Eğer sağda başlarsa düzlüğü geçmesi çok zaman alır. Çok erken durdurursanız, global minimuma asla ulaşamazsınız.

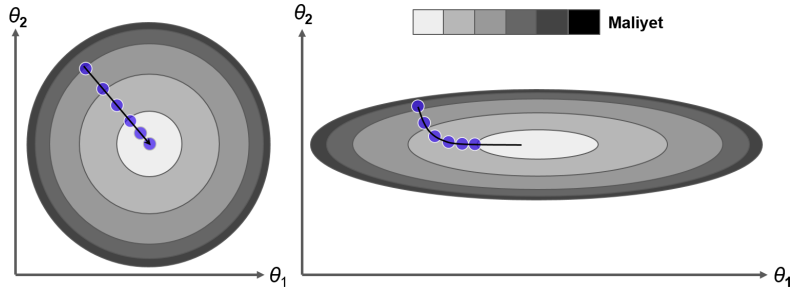
Neyse ki doğrusal bağlanım modeli için MSE maliyet fonksiyonu *konveks fonksiyondur*. Bu, eğri üzerinde seçeceğiniz iki noktadan geçen bir doğru parçasının eğriyi asla kesmeyeceği anlamına gelir. Bu da yerel minimum olmadığını, sadece



Şekil 4.6: *Gradyan inişi tuzağı*

tek olan global minimum olduğunu işaret eder. Aynı zamanda eğimi birden değişmeyen sürekli bir fonksiyondur.[3] Bu iki gerçeğin bir güzel sonucu vardır: Gradyan inişi global minimum değerine yeterince yaklaşacağını garanti eder (yeterince uzun beklerseniz ve öğrenme oranı çok büyük değilse).

Aslında maliyet fonksiyonunun şekli çanak şeklindedir ancak eğer nitelikler farklı ölçeklerdeyse yayvan bir çanak olabilir. Şekil-4.7 eğitim veri setindeki nitelik 1 ve 2'nin aynı ölçeğe geldiğindeki gradyan inişini (soldaki) ve nitelik 1'in nitelik 2'ye göre daha küçük değerler aldığı durumdaki gradyan inişini (sağdaki) gösteriyor.[4]



Şekil 4.7: *Öznitelik ölçeklemesi olan (sol) ve olmayan (sağ) gradyan inişi*

Gördüğünüz gibi soldaki gradyan inişi algoritması doğrudan minimuma gidiyor ve bu minimuma hızlıca ulaşıyor. Ama sağdaki önce global minimum istikametine hemen hemen ortogonal yönde gidiyor ve düz vadide uzun bir yol katediyor. Sonunda minimuma ulaşacaktır ancak bu uzun zaman alacaktır.

[3] Teknik olarak türevi *Lipschitz süreklisidir.*

[4] Nitelik 1 daha küçük değer aldığından, maliyet fonksiyonunu etkileyebilmek için θ_1'de daha büyük bir değişim gereklidir. Bu nedenle, çanak θ_1 ekseni boyunca uzamıştır.

Gradyan inişini kullanırken özniteliklerin aynı ölçekte olduğuna emin olmalısınız (örneğin Scikit-Learn'in `StandardScaler` sınıfını kullanarak) yoksa yakınsaması uzun zaman alacaktır.

Bu diyagram aynı zamanda model eğitmenin maliyet fonksiyonu enküçülten (eğitim veri setinde) parametre kombinasyonu arama anlamına geldiğini gösteriyor. Bu arama, modelin *parametre uzayında* yapılır: Model parametre sayısı arttıkça, bu uzayın boyutları çok artar ve arama zorlaşır: 300-boyutlu samanlıkta iğne aramak 3 boyutta aramaktan daha zorludur. Neyse ki doğrusal bağlanımda maliyet fonksiyonu konvekstir ve iğne de çanağın dibindedir.

4.2.1 Yığın Gradyan İnişi

Gradyan inişini gerçekleştirmek için modelin her parametresi θ_j için maliyet fonksiyonunun gradyanını hesaplamalısınız. Diğer bir deyişle θ_j parametresini bir miktar değiştirdiğinizde maliyet fonksiyonunun ne kadar değişeceğini hesaplamaya ihtiyacınız vardır. Bu, kısmi türev olarak adlandırılır. Şöyle sormaya benzer; "Eğer yüzümü batıya dönersem dağın ayağımın altında kalan kısmının eğimi nedir?" ve aynı soruyu kuzey için de sorarsınız (ve bunu her yön için tekrar edersiniz). Denklem 4.5 maliyet fonksiyonunun θ_j parametresine göre kısmi türevini hesaplar ve $\partial \mathrm{MSE}(\boldsymbol{\theta})/\partial\theta_j$ ile gösterilir.

Maliyet fonksiyonunun kısmi türevi

$$\frac{\partial}{\partial\theta_j}\mathrm{MSE}\left(\theta\right) = \frac{2}{m}\sum_{i=1}^{m}\left(\theta^{\intercal}\mathbf{x}^{(i)} - y^{(i)}\right)x_j^{(i)} \tag{4.5}$$

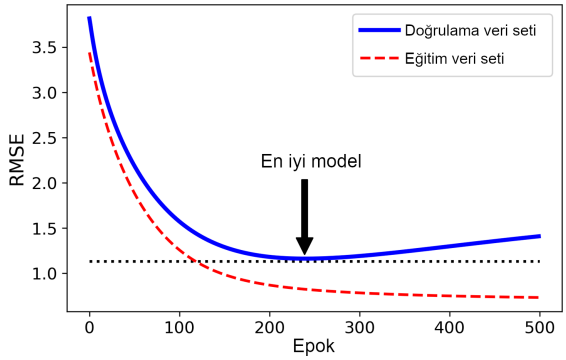
Tek tek hesaplamak yerine Denklem 4.6'daki vektörü kullanarak kısmi türevleri bir seferde de hesaplayabilirsiniz. $\nabla_\theta \mathrm{MSE}(\theta)$ ile gösterilen gradyan vektörü, maliyet fonksiyonunun tüm kısmi türevlerini içerir (her biri, model parametresi için olacak biçimde).

Maliyet fonksiyonunun gradyan vektörü

$$\nabla_\theta\,\mathrm{MSE}\left(\theta\right) = \begin{pmatrix} \frac{\partial}{\partial\theta_0}\mathrm{MSE}\left(\theta\right) \\ \frac{\partial}{\partial\theta_1}\mathrm{MSE}\left(\theta\right) \\ \vdots \\ \frac{\partial}{\partial\theta_n}\mathrm{MSE}\left(\theta\right) \end{pmatrix} = \frac{2}{m}\mathbf{X}^{\intercal}(\mathbf{X}\theta - \mathbf{y}) \tag{4.6}$$

4.5.4 Erken Durdurma

Gradyan inişi gibi iterasyonla çalışan öğrenme algoritmalarını düzenlileştirmenin çok farklı bir yoluysa doğrulama hatası en küçük değere ulaşır ulaşmaz eğitimi durdurmaktır. Bu, *erken durdurma* olarak adlandırılır. Şekil-4.20 yığın gradyan inişiyle eğitilen karmaşık bir modeli (yüksek dereceden polinom bağlanım modeli) göstermektedir. Epoklar ilerledikçe ve algoritma öğrendikçe eğitim ve doğrulama veri setlerinde tahmin hatası (RMSE) birlikte azalır. Bir süre sonra doğrulama hatasının azalması durur ve tekrar yükselişe geçer. Bu, modelin eğitim veri setine aşırı uydurmaya başladığına işaret eder. Erken durdurmada doğrulama hatası en küçük değere iner inmez eğitimi durdurursunuz. Bu, Geoffrey Hinton'ın "bedava öğlen yemeği" dediği, basit ve etkili bir düzenlileştirme tekniğidir.



Şekil 4.20: *Erken durdurma düzenlileştirmesi*

Stokastik ve mini-yığın gradyan inişi kullanırken eğriler bu kadar düzgün olmaz ve bazen minimuma ulaşıp ulaşmadığınızı anlamak zor olabilir. Çözümlerden biri, doğrulama hatasının minimum civarında bir süre geçirmesi hâlinde durmak (modelin daha iyisini yapamayacağına emin olduğunuzda) ve model parametrelerini doğrulama hatasının en küçük olduğu duruma döndürmektir.

Aşağıda erken durdurmanın basit bir uygulamasını bulabilirsiniz:

```
from copy import deepcopy
# verileri hazırlama
poly_scaler = Pipeline([
        ("poly_features", PolynomialFeatures(degree=90, include_bias=False)),
        ("std_scaler", StandardScaler())
    ])
```

```
X_train_poly_scaled = poly_scaler.fit_transform(X_train)
X_val_poly_scaled = poly_scaler.transform(X_val)

sgd_reg = SGDRegressor(max_iter=1, tol=-np.infty, warm_start=True,
                       penalty=None, learning_rate="constant", eta0=0.0005)
minimum_val_error = float("inf")
best_epoch = None
best_model = None

for epoch in range(1000):
    sgd_reg.fit(X_train_poly_scaled, y_train)  # kaldığı yerden devam ediyor
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
    val_error = mean_squared_error(y_val, y_val_predict)
    if val_error < minimum_val_error:
        minimum_val_error = val_error
        best_epoch = epoch
        best_model = deepcopy(sgd_reg)
```

`warm_start=True` seçilip `fit()` metodu çağrıldığında eğitimin sıfırdan başlamak yerine kaldığı yerden devam ettiğini unutmayın.

4.6 Lojistik Bağlanım

Bölüm 1'de aktardığımız gibi bazı bağlanım algoritmaları sınıflandırma için kullanılabilir (tersi de geçerlidir). *Lojistik bağlanım* genellikle örneklerin bir sınıfa ait olma olasılığını tahmin etmek için kullanılır (örneğin, bir e-postanın gereksiz posta olma olasılığı nedir?). Eğer tahmin edilen olasılık %50'den fazlaysa örnek o sınıfa aittir (*pozitif sınıf* denir ve etiketi "1"dir), değilse o sınıfa ait değildir (*negatif sınıfa* aittir ve etiketi "0"dır). Bu, onu ikili sınıflandırıcı yapar.

4.6.1 Olasılık Tahmini

Lojistik bağlanım nasıl çalışır? Doğrusal bağlanım modelinde olduğu gibi lojistik bağlanım modeli de girdi niteliklerinin ağırlıklı toplamını (artı yanlılık terimi) hesaplar. Ama doğrusal bağlanım modelinin yaptığı gibi doğrudan bir çıktı vermek yerine sonucun *lojistiğini* verir (Denklem 4.13'e bakınız).

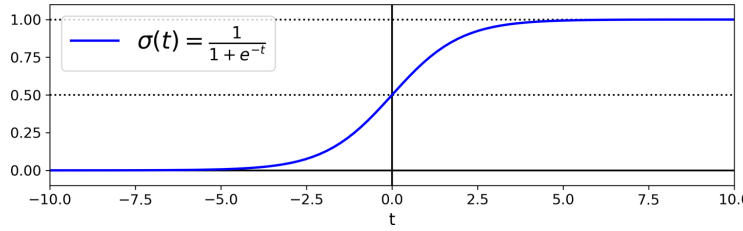
Lojistik —$\sigma(\cdot)$ ile gösterilen— sigmoid fonksiyonudur (S-şeklinde) ve 0 ile 1 arasında çıktılar verir. Denklem 4.14 ve Şekil-4.21'de gösterildiği gibi tanımlanır.

Lojistik bağlanım modeli olasılık tahmini (vektörel form)

$$\widehat{p} = h_\theta(\mathbf{x}) = \sigma(\mathbf{x}^\intercal \theta) \tag{4.13}$$

Lojistik fonksiyon

$$\sigma(t) = \frac{1}{1 + exp(-t)} \tag{4.14}$$



Şekil 4.21: *Lojistik fonksiyon*

Lojistik bağlanım modeli, $\mathbf{x}$ örneğinin pozitif sınıfa ait olma olasılığı olan $(\hat{p}) = h_\theta(\mathbf{x})$'i hesapladığında o örneğe ait $\hat{y}$ tahminini kolaylıkla yapabilir (Denklem 4.15'e bakınız).

Lojistik bağlanım modelinin tahmini

$$\widehat{y} = \begin{cases} 0, & \widehat{p} < 0.5 \\ 1, & \widehat{p} \geq 0.5 \end{cases} \tag{4.15}$$

$t < 0$ olduğunda $\sigma(t) < 0.5$ olduğuna ve $t \geq 0$ olduğunda $\sigma(t) \geq 0.5$ olduğuna dikkat edin. Sonuç olarak lojistik bağlanım modeli $\mathbf{x}^\top \theta$ pozitif olduğunda 1 negatif olduğunda 0 tahmin eder.

Bu t skoru, lojit olarak da adlandırılır. İsmi, lojit(p) = log$(p$ / $(1 - p))$ ile tanımlanan lojit fonksiyonundan gelir. Lojit fonksiyonu, lojistik fonksiyonun tersidir. Tahmin edilen olasılık p'nin lojitini hesapladığınızda sonucun t olduğunu bulacaksınız. Pozitif sınıflar için tahmin edilen olasılık ile negatif sınıflar için tahmin edilen olasılık arasındaki oranın logaritması olduğu için lojit aynı zamanda *odds oranının logaritması* olarak da bilinir.

Lojistik maliyet fonksiyonunun kısmi türevleri

$$\frac{\partial}{\partial \theta_j} \mathrm{J}\left(\theta\right) = \frac{1}{m} \sum_{i=1}^{m} \left(\sigma\left(\theta^{\mathsf{T}} \mathbf{x}^{(i)}\right) - y^{(i)} \right) x_j^{(i)} \tag{4.18}$$

Stokastik gradyan inişinde her seferinde bir örnek ve mini-yığın gradyan inişinde her seferinde bir mini-yığın alırsınız.

4.6.3 Karar Sınırları

Iris veri setini kullanarak lojistik bağlanımı görselleştirelim. Üç farklı türden 150 süsen çiçeğinin çanak ve taç yapraklarının genişliği ve uzunluğunu içeren çok ünlü bir veri setidir: *Iris setosa*, *Iris versicolor* ve *Iris virginica* (Şekil-4.22'ye bakınız[14]).

Şekil 4.22: *Üç farklı süsen bitkisinin çiçekleri*

Şimdi *Iris virginica* türünü sadece taç yaprağı genişliğine bakarak tespit eden bir sınıflandırıcı oluşturalım. Önce veri setini yükleyelim:

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> list(iris.keys())
['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filename']
>>> X = iris["data"][:, 3:]  # taç yaprağı genişliği
>>> y = (iris["target"] == 2).astype(np.int)  # eğer Iris virginica ise 1 değilse 0
```

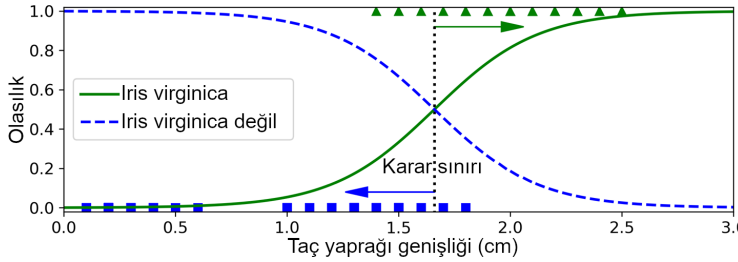
[14] Resimler ilgili Wikipedia sayfaları kullanılarak yeniden üretilmiştir. *Iris virginica* resmi, Frank Mayfield (Creative Commons BY-SA 2.0 *(https://creativecommons.org/licenses/by-sa/2.0/)*), *Iris versicolor* resmi D. Gordon E. Robertson (Creative Commons BY-SA 3.0 *(https://creativecommons.org/licenses/by-sa/3.0/)*), *Iris setosa* resmi açık kaynaklı.

Şimdi de bir lojistik bağlanım modeli eğitelim:

```
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression()
log_reg.fit(X, y)
```

Genişliği 0 cm'den 3 cm'ye değişen taç yaprağa sahip çiçekler için modelin tahmin ettiği olasılıklara bakalım (Şekil-4.23):[15]

```
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)
y_proba = log_reg.predict_proba(X_new)
plt.plot(X_new, y_proba[:, 1], "g-", label="Iris virginica")
plt.plot(X_new, y_proba[:, 0], "b--", label="Not Iris virginica")
# + güzel gözükmesi için daha fazla Matplotlib kodu
```



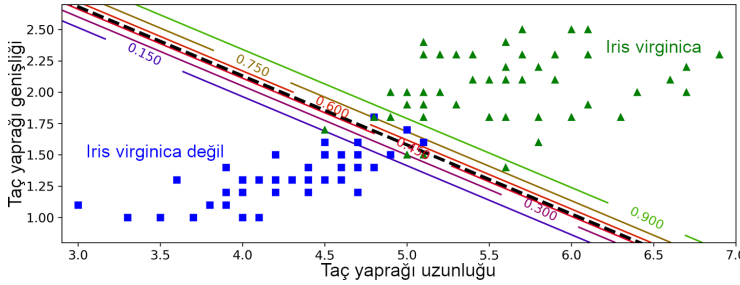
Şekil 4.23: *Tahmin edilen olasılıklar ve karar sınırı*

Iris virginica çiçeğinin taç yapraklarının genişliği (üçgenle gösterilen) 1.4 cm ile 2.5 cm aralığındadır, diğer süsen çiçeklerinin taç yaprağı genişliğiyse (kareyle gösterilenler) daha küçüktür 0.1 cm ile 1.8 cm aralığındadır. Bir miktar üst üste geldiğine dikkat edin. Taç yaprağı genişliği 2 cm'nin üstüne çıktığında sınıflandırıcı çiçeğin *Iris virginica* (o sınıf için yüksek olasılık değerleri veriyor) olduğuna oldukça eminken, 1 cm'in altına indiğinde *Iris virginica* olmadığına oldukça emindir. Bu iki ucun haricinde sınıflandırıcı kararsız kalıyor ancak predict sınıfını (`predict_proba()` metodu yerine `predict()` metodu) kullanarak bir sınıf tahmin etmesini isterseniz, en olası sınıfı getirecektir. Dolayısıyla 1.6 cm civarında olasılık değerlerinin %50-%50 eşit olduğu bir karar sınırı bulunuyor. Eğer taç yaprağı genişliği 1.6 cm'den büyükse sınıflandırıcı çiçeği *Iris virginica* olarak sınıflandıracak aksi hâlde de *Iris virginica* olmadığını tahmin edecek (her ne kadar çok emin olmasa da):

```
>>> log_reg.predict([[1.7], [1.5]])
array([1, 0])
```

[15] NumPy'ın `reshape()` fonksiyonu bir boyutun –1 olmasına izin verir, bu "tanımlanmamış" anlamındadır: Bu değer dizinin uzunluğundan ve geri kalan boyutlardan elde edilir.

Şekil-4.24 aynı veri setini ancak bu kez iki niteliğini gösteriyor: Taç yaprak uzunluğu ve genişliği. Lojistik bağlanım sınıflandırıcı eğitildikten sonra bu iki niteliği kullanarak bir çiçeğin *Iris virginica* olup olmadığını tahmin edebilir. Kesikli çizgi modelin %50 olasılığı tahmin ettiği noktaları göstermektedir: Bu da modelin karar sınırıdır. Doğrusal bir sınır olduğuna dikkat edin.[16] Her paralel doğru modelin %15'ten (sol alt) %90'a (sağ üst) farklı olasılık değeri verdiği noktaları temsil etmektedir. Modele göre sağ üstteki doğrunun ötesinde bulunan bir çiçek %90'dan fazla ihtimalle Iris virginica olarak sınıflandırılacaktır.



Şekil 4.24: *Doğrusal karar sınırı*

Diğer doğrusal modeller gibi lojistik bağlanım modeli ℓ_1 ya da ℓ_2 kullanarak düzenlileştirilebilir. Scikit-Learn varsayılan olarak ℓ_2 kullanır.

Scikit-Learn `LogisticRegression` modeli için düzenlileştirme seviyesinin kontrolü `alpha` (diğer doğrusal modeller de olduğu gibi) parametresi yerine bu parametrelerin tersini kullanarak yapılır: `C`. `C`'nin değeri yükseldikçe model daha *az* düzenlileştirilir.

4.6.4 Softmax Bağlanım

Lojistik bağlanım modeli çok sınıflılığı destekleyecek şekilde birden fazla ikili sınıflandırıcı eğitmeye gerek olmaksızın doğrudan genelleştirilebilir. (Bölüm 3'te anlatıldığı gibi). Bu, *softmax bağlanım* ya da *çok terimli lojistik bağlanım* olarak adlandırılır.

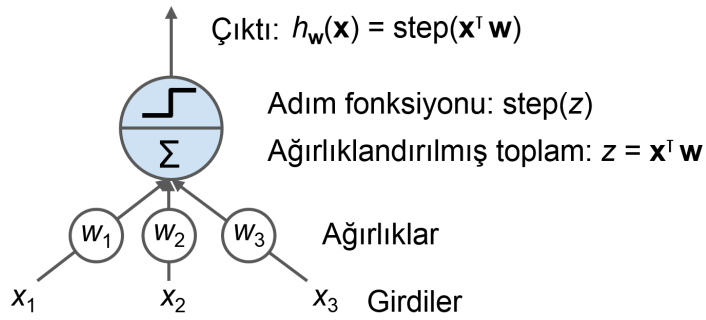
Fikir çok basit: Verilen bir $\mathbf{x}$ örneği için softmax bağlanım modeli önce her k sınıfı için bir skor $s_k(\mathbf{x})$ hesaplar, sonra da bu skorlara *softmax fonksiyonunu* (normalleştirilmiş üstel fonksiyon da denir) uygulayarak her sınıfın olasılığını tahmin eder. $s_k(\mathbf{x})$'yı hesaplamak için kullanılan denklem doğrusal bağlanımın tahminlerini yaptığı denkleme benzetilebilir (Denklem 4.19'a bakınız).

[16] $\mathbf{x}$ nokta kümesi $\theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0$ ile tanımlanır ve düz bir doğrudur.

Bu sinir ağlarının karmaşık mantıksal ifadeleri hesaplamak üzere nasıl bir araya getirilebileceğini hayal edebilirsiniz (örnek için bölüm sonundaki alıştırmalara bakınız).

10.1.3 Perseptron (Algılayıcı)

Perseptron, 1957'de Frank Rosenblatt tarafından icat edilmiş olan en basit ANN mimarilerinden biridir. Eşik mantık birimi (TLU: *Threshold Logic Unit*) veya bazen doğrusal eşik birimi de (LTU: *Linear Threshold Unit*) denilen biraz farklı bir yapay sinir hücresine dayanır (Şekil-10.4'e bakınız). Girdiler ve çıktılar sayılardır (ikili açık/kapalı değerleri yerine) ve her bir girdi bağlantısı bir ağırlık değeriyle ilişkilidir. TLU girdilerin ağırlıklandırılmış toplamını hesaplar ($z = w_1\ x_1 + w_2\ x_2 + \cdots + w_n\ x_n = \mathbf{x}^\intercal\ \mathbf{w}$) ve daha sonra bu toplama adım fonksiyonu uygular ve sonucu çıkarır: $h_{\mathbf{w}}(\mathbf{x}) = \text{step}(z)$, burada $z = \mathbf{x}^\intercal\ \mathbf{w}$'dır.



Şekil 10.4: *Eşik mantık birimi: girdilerinin ağırlıklandırılmış toplamını hesaplayan ve daha sonra adım fonksiyonu uygulayan yapay bir sinir hücresi*

Perseptronlarda en yaygın kullanılan adım fonksiyonu Heaviside adım fonksiyonudur (Denklem 10.1'e bakınız). Bunun yerine bazen işaret fonksiyonu (sgn) da kullanılır.

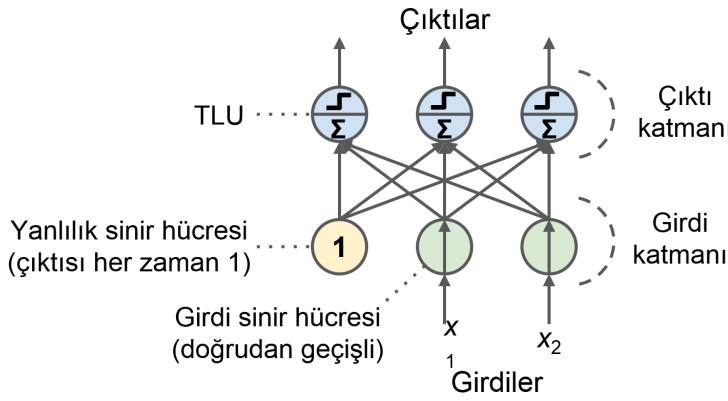
Perseptronlarda yaygın olarak kullanılan adım fonksiyonları (eşik=0 olduğunu *varsayarak*)

$$heaviside\,(z) = \begin{cases} 0, & z < 0 \\ 1, & z \geq 0 \end{cases} \qquad sgn\,(z) = \begin{cases} -1, & z < 0 \\ 0, & z = 0 \\ +1, & z > 0 \end{cases} \tag{10.1}$$

Tek bir TLU basit doğrusal sınıflandırma için kullanılabilir. Girdilerin doğrusal bir bileşimini hesaplar ve sonuç belirli bir eşiği aştığında pozitif sınıfı sonucunu verir. Aksi takdirde negatif sınıf çıkarır (doğrusal bağlanım veya doğrusal

SVM sınıflandırıcısı gibi). Örneğin, taç yaprak uzunluğu ve genişliğine göre Süsen çiçeğini sınıflandırmak için tek bir TLU kullanabilirsiniz (daha önceki bölümlerde yaptığımız gibi ilave yanlılık özniteliği olarak $x_0 = 1$ ekleyerek). Bu durumda TLU'yu eğitmek w_0, w_1, ve w_2 için doğru değerleri bulmak anlamına gelir (eğitim algoritmasından birazdan bahsedeceğiz).

Bir perseptron her bir TLU bütün girdilere bağlanacak şekilde TLU'ların[7] tek bir katmanından oluşur. Bütün sinir hücreleri bir önceki katmandaki (yani girdi sinir hücreleri) her bir sinir hücresine bağlandığında bu katmana *tam bağlantılı katman* veya *yoğun katman* (ÇN: Dense layer) denir. Perseptronun girdileri *girdi sinir hücreleri* denilen doğrudan geçişli sinir hücrelerine beslenir: Hangi girdi ile beslenmişlerse onu çıktı olarak verirler. Bütün girdi sinir hücreleri *girdi katmanını* oluşturur. Dahası, genelde ilave bir yanlılık özniteliği eklenir ($x_0 = 1$): Tipik olarak *yanlılık sinir hücresi* denilen özel bir sinir hücresi çeşidiyle temsil edilirler. Bu sinir hücresinin çıktısı her zaman 1'dir. İki girdili ve üç çıktılı bir Perseptron Şekil-10.5'te gösterilmiştir. Bu Perseptron, örnekleri eş zamanlı olarak üç farklı ikili sınıfa sınıflandırabilir. Bu, Perseptronu çok etiketli bir sınıflandırıcı yapar.



Şekil 10.5: *İki girdi sinir hücresi, bir yanlılık sinir hücresi ve üç çıktı sinir hücresi olan bir Perseptronun mimarisi*

Doğrusal cebirin sihri sayesinde, Denklem 10.2, yapay sinir hücreleri katmanının çıktılarını birkaç örnek için tek seferde etkin bir şekilde hesaplamayı mümkün kılar.

Yoğun katmanın çıktılarını hesaplamak

$$h_{\mathbf{W},\mathbf{b}}(\mathbf{X}) = \phi(\mathbf{X}\mathbf{W} + \mathbf{b}) \tag{10.2}$$

[7] *Perseptron* ismi bazen tek TLU'lu küçük bir ağ manasında da kullanılır.

Bu eşitlikte:

- Her zamanki gibi **X**, girdi özniteliklerinin matrisini temsil eder. Her bir örnek için bir satırı ve her bir öznitelik için bir sütunu vardır.
- Ağırlık matrisi **W**, yanlılık sinir hücresinden olanlar haricinde bütün bağlantı ağırlıklarını içerir. Katmandaki her bir girdi sinir hücresi için bir satırı ve her bir yapay sinir hücresi için bir sütunu vardır.
- Yanlılık vektörü **b**, yanlılık sinir hücresi ve yapay sinir hücreleri arasındaki bütün bağlantı ağırlıklarını içerir. Her bir yapay sinir hücresi için bir yanlılık terimi vardır.
- ϕ fonksiyonuna *aktivasyon fonksiyonu* denir: Yapay sinir hücreleri TLU'lar olduğunda, bu aktivasyon fonksiyonu bir adım fonksiyonudur (aktivasyon fonksiyonlarından birazdan bahsedeceğiz).

Peki bir Perseptron nasıl eğitilir? Rosenblatt tarafından öne sürülen Perseptron eğitim algoritması çoğunlukla Hebb kuralından etkilenmişti. 1949 tarihli *The Organization of Behavior* isimli kitabında Donald Hebb, bir biyolojik sinir hücresinin diğer bir sinir hücresini sıkça tetiklediği zaman bu iki sinir hücresinin arasındaki bağlantının daha da güçlendiğini iddia etmiştir. Siegrid Löwel daha sonra Hebb'in fikrini hatırlaması kolay olan şu cümleyle özetledi: "Birlikte ateşlenen hücreler birbirine bağlanır", yani, iki sinir hücresi arasındaki bağlantının ağırlığı hücreler eş zamanlı olarak ateşlendiğinde artma eğilimi gösterir. Bu kural daha sonra Hebb's kuralı (veya *Hebbian öğrenmesi*) olarak anıldı. Perseptronlar, tahmin yaparken, ağ tarafından yapılan hatayı hesaba katan bu kuralın bir çeşiti kullanılarak eğitilir. Perseptron öğrenme kuralı bağlantıları pekiştirir ve bu da hatayı azaltmaya yardımcı olur. Daha spesifik olarak, Perseptron her seferinde bir eğitim örneğiyle beslenir ve her bir örnek için tahminlerde bulunur. Yanlış tahminde bulunan her sinir hücresi için doğru tahmine katkıda bulunmuş olabilecek girdilerden gelen bağlantı ağırlıklarını pekiştirir. Kural, Denklem 10.3'te gösterilmiştir.

Perseptron öğrenme kuralı (ağırlık güncelleme)

$$w_{i,j}{}^{(\text{sonraki adım})} = w_{i,j} + \eta(y_j - \widehat{y}_j)x_i \tag{10.3}$$

Bu eşitlikte:

- $w_{i,\,j}$, i^{inci} girdi sinir hücresi ve j^{inci} çıktı sinir hücresi arasındaki bağlantı ağırlığıdır.

- x_i, mevcut eğitim örneğinin i^{inci} girdi değeridir.
- $(\hat{y}_j)$, mevcut eğitim örneği için j^{inci} çıktı sinir hücresinin çıktısıdır.
- y_j, mevcut eğitim örneği için j^{inci} çıktı sinir hücresinin hedef çıktısıdır.
- η öğrenme oranıdır.

Her bir çıktı sinir hücresinin karar sınırı doğrusaldır dolayısıyla Perseptronlar karmaşık yapıları nedeniyle öğrenmede yetersizdirler (lojistik bağlanım sınıflandırıcıları gibi). Bununla birlikte, Rosenblatt, eğitim örneklerinin doğrusal ayrılabilir olması durumunda bu algoritmanın bir çözüme yakınsayacağını göstermiştir.[8] Buna *Perseptron yakınsama teoremi* denir.

Scikit-Learn tek bir TLU ağı gerçekleyen bir `Perceptron` sınıfı sağlamaktadır. Hemen hemen tahmin edeceğiniz şekilde kullanılabilir. Örneğin iris veri setinde (Bölüm 4'te gösterilmiştir).

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron
iris = load_iris()
X = iris.data[:, (2, 3)] # taç yaprak uzunluğu, taç yaprak genişliği
y = (iris.target == 0).astype(np.int) # Iris setosa?
per_clf = Perceptron()
per_clf.fit(X, y)
y_pred = per_clf.predict([[2, 0.5]])
```

Perseptron öğrenme algoritmasının stokastik gradyan inişine çok benzediği dikkatinizi çekmiştir. Aslında Scikit-Learn'nün `Perceptron` sınıfı şu hiperparametrelerle `SGDClassifier` kullanmaya denktir: `loss=''perceptron''`, `learning_rate=''constant`, `eta0=1` (öğrenme oranı) ve `penalty=None` (düzenlileştirme yok).

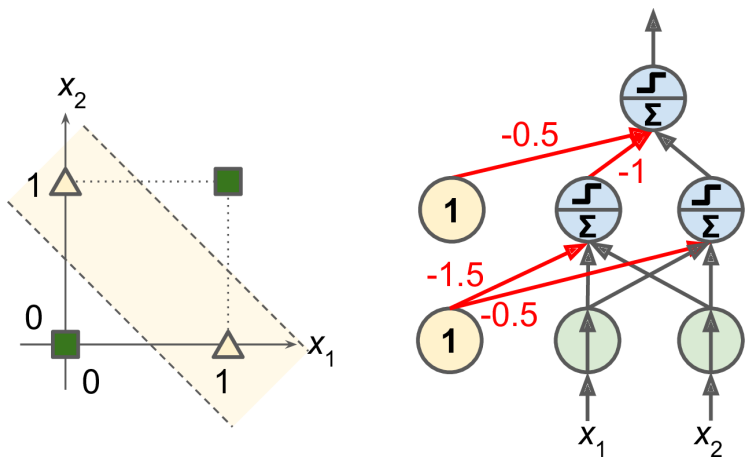
Lojistik bağlanım sınıflandırıcılarının aksine Perseptronların çıktı olarak sınıf olasılığı vermediğine dikkat edin; daha ziyade sert bir eşiğe dayalı olarak tahminde bulunurlar. Lojistik bağlanımı Perseptronlara nazaran tercih etme nedenlerinden biri budur.

1969 tarihli *Perseptronlar* adlı monografta Marvin Minsky ve Seymour Papert Perseptronların bir dizi ciddi zayıflığının altını çizdiler —özellikle bazı basit problemleri çözmede yetersiz olmaları (*örneğin dışlayıcı* VEYA (XOR)

[8] Bu çözümün emsalsiz olmadığına dikkat edin: Veri noktaları ayrılabilir olduğunda onları ayırabilecek sonsuz sayıda hiperdüzlem bulunur.

sınıflandırma problemi; Şekil-10.6'nın sol tarafına bakınız). Bu, diğer doğrusal sınıflandırma modelleri için de geçerlidir (Lojistik Bağlanım sınıflandırıcısı gibi) ancak araştırmacılar Perseptronlardan daha fazlasını ümit ettiler ve bazıları öyle büyük hayal kırıklığına uğramışlardı ki mantık, problem çözme ve arama gibi daha üst düzey sorunlar uğruna sinir ağından tümden vazgeçtiler.

Öyle görünüyor ki Perseptronların kısıtlarının bazıları çoklu Perseptronları üst üste yığarak giderilebilir. Oluşan ANN'ye *çok katmanlı Perseptron* (MLP: Multilayer Perceptron) denir. MLP, XOR problemini çözebilir. Bunu Şekil-10.6'nın sağ tarafında gösterilen MLP'nin çıktısını hesaplayarak doğrulayabilirsiniz: (0,0) veya (1,1) girdileri ile ağ 0 çıktısını verirken (0,1) ve (1,0) girdileri ile 1 çıktısını verir. Ağırlığı gösterilen dört bağlantı haricinde bütün bağlantıların ağırlığı 1'dir. Bu ağın XOR problemini gerçekten çözdüğünü doğrulamaya çalışınız!



Şekil 10.6: *XOR sınıflandırma problemi ve onu çözen bir MLP*

10.1.4 Çok Katmanlı Perseptron ve Geri Yayılım

Bir MLP, bir adet (doğrudan geçişli) *girdi katmanından*, *gizli katmanlar* adı da verilen bir veya daha fazla TLU'lar içeren katmanlardan ve *çıktı katmanı* (Şekil-10.7'ye bakınız) denilen son bir TLU'lar katmanından oluşur. Girdi katmanına yakın olan katmanlar genellikle *alt katmanlar* ve çıktı katmanına yakın olanlar ise genellikle *üst katmanlar* olarak adlandırılır. Çıktı katmanı hariç her katman bir yanlılık sinir hücresi içerir ve bu yanlılık sinir hücresi bir sonraki katmana tam bağlıdır.

16.4.1 Görsel İlgi

İlgi mekanizmaları şimdilerde çeşitli amaçlar için kullanılmaktadır. NMT'nin ötesindeki ilk uygulamalarından biri görsel ilgi (*https://homl.info/visualattention*)[17] kullanarak görüntü başlığı üretmekteydi: Evrişimsel bir sinir ağı önce görüntüyü işler ve bazı öznitelik haritalarını çıktı olarak verir. Daha sonra ilgi mekanizmasıyla donatılmış bir kodçözücü RNN her seferinde bir kelime olmak üzere başlığı üretir. Her bir kodçözücü zaman adımında (her bir kelime), kodçözücü, görüntünün doğru kısmına odaklanmak için ilgi modelini kullanır. Örneğin Şekil-16.7'de[18] model "A woman is throwing a frisbee in a park" (ÇN: Bir kadın parkta frizbi atıyor) başlığını üretmiştir. "frisbee" (ÇN: Frizbi) kelimesini çıktı olarak vermek üzereyken kodçözücünün görüntünün hangi kısmına ilgisini odakladığını görebilmektesiniz: Açık bir şekilde ilgisinin çoğu frizbi üzerine odaklanmıştır.

Şekil 16.7: *Görsel ilgi: Bir resim girdisi (solda) ve modelin "frisbee" kelimesini üretmeden önce odaklanması (sağda)*

Açıklanabilirlik

İlgi mekanizmalarının ilave bir faydası, modelin çıktısının üretilmesine neyin yol açtığını anlamayı kolaylaştırmalarıdır. Buna *açıklanabilirlik* denir. Özellikle de model bir hata yaptığında kullanışlıdır: Örneğin, karda yürüyen bir köpeğin görüntüsü için "a wolf walking in the snow" (ÇN: Karda yürüyen bir kurt) şeklinde bir başlık verilmişse geri gidip

[17] Kelvin Xu et al., "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention," *Proceedings of the 32nd International Conference on Machine Learning* (2015): 2048–2057.

[18] Bu, makaledeki Şekil 3'ün bir kısmıdır. Yazarların kibar izniyle yeniden üretilmiştir.

> modelin "wolf" (ÇN: Kurt) kelimesini çıktı olarak verdiğinde neye odaklandığını inceleyebilirsiniz. Sadece köpeğe ilgi göstermediğini aynı zamanda kara da ilgi gösterdiğini görebilirsiniz. Bu ise olası bir açıklamayı işaret eder: Belki de modelin köpekleri kurtlardan ayırt etmeyi öğrenme yöntemi etrafta çok kar olup olmadığını kontrol etmekle olmaktadır. Böylece bu hatayı modeli etrafta kar olmayan daha fazla kurt görüntüleri ve etrafta kar olan köpek görüntüleriyle eğiterek giderebilirsiniz. Bu örnek, Marco Tulio Riberio vd. tarafından yazılan, açıklanabilirliğe yönelik daha farklı yaklaşım kullanan 2016 yılına ait harika bir makaleden (*https://homl.info/explainclass*)[19] gelmektedir: Bir sınıflandırıcının tahmini etrafında yerel olarak yorumlanabilir bir modeli öğrenmek.
>
> Bazı uygulamalarda açıklanabilirlik sadece bir modelde hata ayıklama aracı değildir; aynı zamanda yasal bir gereklilik de olabilir (size kredi verip vermeyeceğine karar veren bir sistemi düşünün).

İlgi mekanizmaları o kadar güçlüdür ki sadece ilgi mekanizmaları kullanarak çok gelişmiş modeller inşa edebilirsiniz.

16.4.2 Tek İhtiyacınız Olan İlgidir: Dönüştürücü Mimarisi

2017 yılına ait çığır açan bir makalede (*https://homl.info/transformer*)[20] Google araştırmacıları "Tek ihtiyacımız olanın ilgi" olduğunu öne sürmüşlerdir. *Dönüştürücü* (ÇN: Transformes) denilen bir mimariyi oluşturmayı başarmışlardır. Bu mimari yinelemeli veya evrişimsel katmanlar[21] kullanmadan sadece ilgi mekanizmaları (artı gömülme katmanları, yoğun katmanlar, normalleştirme katmanları ve birkaç ufak tefek şey) kullanarak NMT'deki teknolojiyi önemli ölçüde iyileştirmiştir. İlave bir bonus olarak, bu mimariyi eğitmesi çok daha hızlıydı ve paralelleştirmesi çok daha basitti, dolayısıyla daha önceki son teknoloji modellerin zaman ve maliyetinin çok altında bu mimariyi eğitmeyi başardılar.

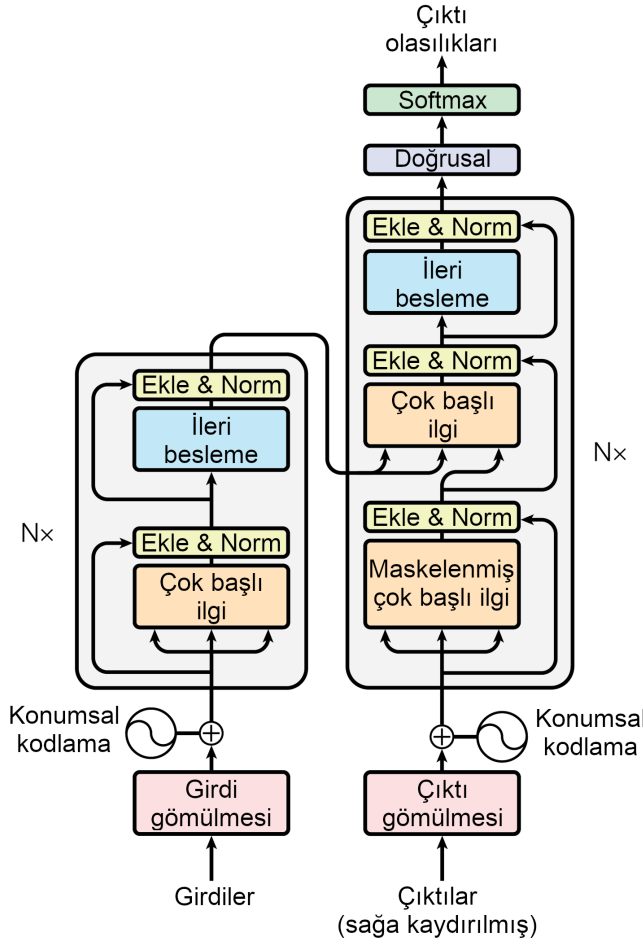
Dönüştürücü mimarisi Şekil-16.8'de[22] gösterilmiştir.

[19] Marco Tulio Ribeiro et al., "'Why Should I Trust You?': Explaining the Predictions of Any Classifier," *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2016): 1135–1144.

[20] Ashish Vaswani et al., "Attention Is All You Need," *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017): 6000–6010.

[21] Dönüştürücü, zaman dağılımlı Dense katmanları kullandığından, kernel büyüklüğü 1 olan bir boyutlu evrişimsel katmanlar kullandığını iddia edebilirsiniz.

[22] Bu, makaledeki Şekil 1'dir. Yazarların kibar izniyle yeniden üretilmiştir.



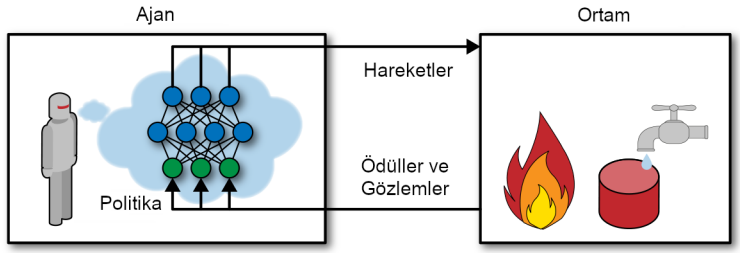
Şekil 16.8: *Dönüştürücü mimarisi*

Bu şekili inceleyelim:

- Sol taraf kodlayıcıdır. Daha önce olduğu gibi kelime numaralarının bir dizisi olarak temsil edilen cümlelerin bir yığınını girdi olarak alır (girdi şekli [*yığın büyüklüğü, en büyük cümle girdisi uzunluğu*]'dur) ve her bir kelimeyi 512 boyutlu bir gösterime kodlar (dolayısıyla kodlayıcının çıktı şekli [*yığın büyüklüğü, en büyük cümle girdisi uzunluğu,* 512]'dir). Kodlayıcının üst kısmının N kere üst üste yığıldığına dikkat ediniz (makalede N=6'dır).
- Sağ taraf kodçözücüdür. Eğitim esnasında hedef cümleyi bir zaman adımı kadar sağa kaydırılmış olarak (en başa dizi başlangıcı andıcı eklenir) girdi alır (kelime numaralarının bir dizisi olarak da temsil edilir). Aynı

18.2 Politika Araması

Bir yazılım ajanının hareketlerini belirlemek için kullandığı algoritmaya o ajanın politikası denir. İlke, gözlemleri girdi olarak alan ve yapılması gereken hareketleri çıktı olarak veren bir sinir ağı olabilir (bakınız Şekil-18.2).



Şekil 18.2: *Bir sinir ağı politikası kullanan pekiştirmeli öğrenme*

İlke düşünebileceğiniz herhangi bir algoritma olabilir ve deterministik olmak zorunda değildir. Aslında bazı durumlarda ortamı gözlemlemek zorunda bile değildir! Örneğin, ödülü 30 dakikada topladığı toz miktarı olan robotik bir elektrikli süpürge düşünün. Bu süpürgenin politikası, her saniye p olasılığıyla ilerlemek veya 1-p olasılığıyla rastgele sola sağa dönmek olabilir. Dönme açısı -r ve +r arasında rastgele bir açı olur. Bu politika bir miktar rastgelelik içerdiğinden *stokastik politika* olarak adlandırılır. Robot düzensiz bir yol takip edecektir. Bu ise ulaşabileceği her yere nihayetinde gideceğini ve bütün tozu toplayacağını garanti eder. Soru ise 30 dakikada ne kadar toz toplayacağıdır.

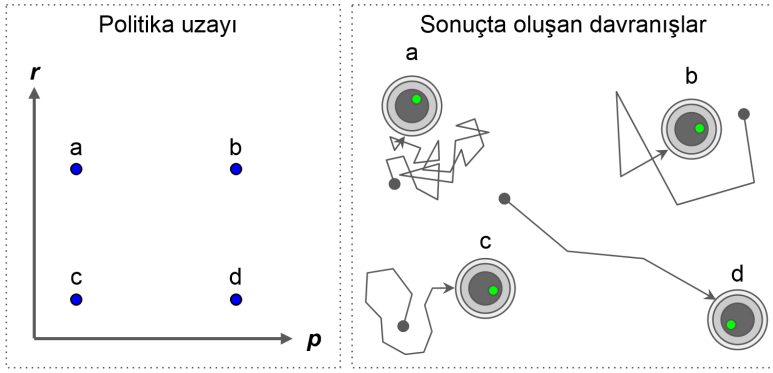
Böyle bir robotu nasıl eğitirsiniz? Ayarlayabileceğiniz sadece iki politika parametresi vardır: p olasılığı ve r açı aralığı. Muhtemel bir öğrenme algoritması bu parametreler için pek çok farklı değer denemek ve en iyi performansı veren kombinasyonu seçmektir (bakınız Şekil-18.3). Bu, *politika aramasının* bir örneğidir. Bu durumda brüt kuvvet yaklaşımı kullanılmaktadır. *Politika uzayı* çok geniş olduğunda (genelde böyledir), bu şekilde iyi bir parametre kümesi bulmak devasa bir saman yığınında iğne aramaya benzer.

Politika uzayını keşfetmenin bir başka yolu *genetik algoritmalar* kullanmaktır. Örneğin, 100 politikanın rastgele ilk neslini oluşturabilir ve bunları deneyebilir, daha sonra en kötü 80 politikayı[6] öldürüp, kurtulan 20 politikanın her birinin 4 yavru vermesini sağlayabilirsiniz. Yavru, ebeveyninin[7] bir kopyası artı bir

[6] "gen havuzunda" bir miktar çeşitliliği muhafaza etmek için zayıf performans gösterenlere azıcık kurtulma şansı vermek sıklıkla iyidir.

[7] Tek bir ebeveyn varsa buna *eşeysiz üreme* denir. İki (veya daha fazla) ebeveyn varsa buna *eşeyli üreme* denir. Yavrunun genomu (bu durumda politika parametreleri kümesi) ebeveynlerinin genomlarının rastgele kısımlarından oluşur.

miktar rastgele değişimdir. Kurtulan politikalar artı yavruları birlikte ikinci nesli oluşturur. İyi bir politika[8] bulana kadar bu şekilde nesiller boyunca döngü kurabilirsiniz.



Şekil 18.3: *Politika uzayında dört nokta (solda) ve ajanın ilgili davranışı (sağda)*

Bir başka yaklaşım, politika parametrelerine göre ödüllerin gradyanlarını hesaplamak ve daha sonra bu parametreleri daha yüksek ödüllere[9] doğru olan gradyanları takip ederek ayarlamak suretiyle eniyileme teknikleri kullanmaktır. *Politika gradyanları* (PG: Policy Gradients) denilen bu yaklaşımdan bu bölümde daha sonra daha ayrıntılı olarak bahsedeceğiz. Elektrikli süpürge robotuna geri dönersek, p'yi az miktarda artırabilir ve bunu yapmanın 30 dakika içinde robot tarafından toplanan toz miktarını artırıp artırmadığını hesaplayabilirsiniz. Eğer artırırsa p'yi biraz daha artırın, aksi takdirde p'yi azaltın. TensorFlow kullanarak popüler bir PG algoritması gerçekleyeceğiz ama bunu yapmadan önce ajanın yaşayacağı bir ortam oluşturmamız gerekiyor —dolayısıyla OpenAI Gym'i tanıtma vakti gelmiştir.

18.3 OpenAI Gym'e Giriş

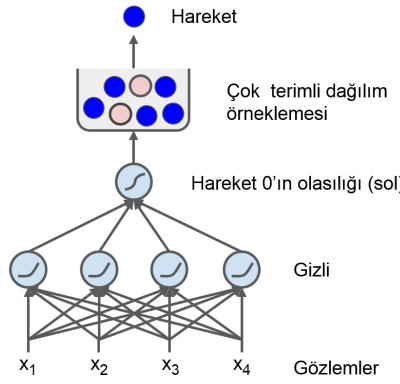
Pekiştirmeli Öğrenmenin zorluklarından biri bir ajanı eğitmek için öncelikle çalışan bir ortamın gerekli olmasıdır. Atari oyununu oynamasını öğrenecek olan bir ajan programlamak istiyorsanız Atari oyun simülatörüne ihtiyacınız olacaktır. Yürüyen bir robot programlamak isterseniz, ortam gerçek dünyadır ve robotunuzu doğrudan bu ortamda eğitebilirsiniz ancak bunun kendi sınırları

[8] Pekiştirmeli Öğrenme için kullanılan ilginç bir genetik algoritma örneği *NeuroEvolution of Augmenting Topologies* (*https://homl.info/neat*) (NEAT) algoritmasıdır.

[9] Buna gradyan çıkışı denir. Gradyan inişi gibidir ama tersi yönde: Enküçültmek yerine enbüyütmek.

18.4 Sinir Ağı Politikaları

Bir sinir ağı politikası oluşturalım. Daha önce gömülü olarak kodladığımız politika gibi bu sinir ağı bir gözlemi girdi olarak alacak ve yerine getirilecek hareketi çıktı olarak verecektir. Daha kesin olmak gerekirse bu sinir ağı her bir hareket için bir olasılık kestirecek ve daha sonra kestirilen olasılıklara göre rastgele bir hareket seçeceğiz (bakınız Şekil-18.5). CartPole ortamı durumunda sadece iki muhtemel hareket vardır (sol veya sağ), dolayısıyla sadece bir tane çıktı sinir hücresine ihtiyacımız vardır. Bu çıktı hareket 0'ın (sol) p olasılığını çıktı olarak verecektir ve elbette ki hareket 1'in (sağ) olasılığı 1-p olacaktır. Örneğin çıktı olarak 0.7'yi verirse hareket 0'ı %70 olasılıkla veya hareket 1'i %30 olasılıkla seçeceğiz demektir.



Şekil 18.5: *5 Sinir ağı politikası*

En yüksek skorlu hareketi seçmekten ziyade neden sinir ağı tarafından verilen olasılıklara dayalı olarak rastgele bir hareket seçtiğimizi merak ediyor olabilirsiniz. Bu yaklaşım ajanın yeni hareketleri *keşfetmek* ile iyi çalıştığı bilinen hareketlerden *faydalanmak* arasındaki doğru dengeyi bulmasını sağlar. İşte bir benzerlik: Bir restorana ilk defa gittiğinizi varsayın. Bütün yemekler aynı güzellikte görünsün. Dolayısıyla rastgele bir tanesini seçersiniz. İyi çıkarsa, bir sonraki sefer bu yemeği isteme olasılığınızı artırabilirsiniz. Ancak bu olasılığı %100'e kadar artırmamalısınız, aksi takdirde diğer yemekleri denemeyebilirsiniz ve bu yemeklerden bazıları denediğinizden daha güzel olabilir.

Her bir gözlem ortamın tam durumunu içerdiğinden, bu özel ortamda geçmiş hareketlerin ve gözlemlerin güvenli bir şekilde ihmal edilebileceğine dikkat ediniz. Birtakım gizli durumlar varsa o zaman geçmiş hareketleri ve gözlemleri göz önüne alabilirsiniz. Örneğin, ortam el arabasının sadece konumunu açığa vurmuşsa ama hızını açığa vurmamışsa o anki hızı kestirmek için sadece o anki gözlemi değil aynı zamanda bir önceki gözlemi de göz önünde bulundurmak

zorunda kalırsınız. Bir başka örnek ise gözlemlerin gürültülü olduğu durumdur. Bu durumda genellikle en muhtemel o anki durumu kestirmek için geçmiş birkaç gözlemleri kullanmak istersiniz. CartPole problemi olabileceği kadar basittir; gözlemler gürültüsüz ve ortamın tam durumunu içermektedir.

Aşağıda tf.keras kullanarak bu sinir ağı politikasını inşa etmek için gerekli kod bulunmaktadır:

```
import tensorflow as tf
from tensorflow import keras

n_inputs = 4 # == env.observation_space.shape[0]

model = keras.models.Sequential([
    keras.layers.Dense(5, activation="elu", input_shape=[n_inputs]),
    keras.layers.Dense(1, activation="sigmoid"),
])
```

`import`'lardan sonra politika ağını tanımlamak için basit bir `Sequential` model kullanıyoruz. Girdi sayısı gözlem uzayının boyudur (CartPole için bu sayı 4'tür) ve sadece beş gizli birimimiz vardır çünkü bu basit bir problemdir. Son olarak, tek bir olasılığı (sola gitme olasılığı) çıktı olarak vermek istiyoruz, dolayısıyla sigmoid aktivasyon fonksiyonu kullanan tek bir çıktı sinir hücremiz vardır. İki muhtemel hareketten daha fazlası olsaydı hareket başına bir çıktı sinir hücresi olurdu ve bu durumda softmax aktivasyon fonksiyonunu kullanırdık.

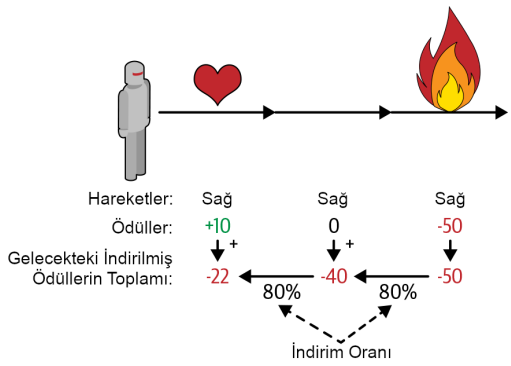
Tamam, şimdi gözlemleri alıp hareket olasılıklarını çıktı olarak verecek bir sinir ağı politikamız var. Ancak bunu nasıl eğitiriz?

18.5 Hareketleri Değerlendirmek: Kredi Atama Problemi

Her adımda en iyi hareketin ne olduğunu bilseydik, kestirilmiş olasılık dağılımı ve hedef olasılık dağılımı arasındaki çapraz entropiyi enküçülterek sinir ağını her zamanki gibi eğitebilirdik. Bu, sıradan denetimli öğrenme olurdu. Ancak pekiştirmeli öğrenmede ajanın aldığı tek kılavuzluk ödüller yoluyladır ve ödüller tipik olarak seyrek ve gecikmelidir. Örneğin ajan, direği 100 adım boyunca dengelemeyi başarırsa, yaptığı 100 hareketin hangilerinin iyi hangilerinin kötü olduğunu nasıl bilecektir? Tek bildiği son hareketten sonra direğin düştüğüdür fakat bu son hareket elbette tamamen sorumlu değildir. Bu, kredi atama problemi olarak bilinir: Ajan bir ödül aldığında hangi hareketlerinin kredilen-

dirildiğini (veya suçlanacağını) bilmesi zordur. İyi davrandıktan saatler sonra ödüllendirilen bir köpek düşünün; niçin ödüllendirildiğini anlayacak mıdır?

Bu problemin üstesinden gelmek için sıkça kullanılan bir strateji, genellikle her adımda bir γ (gama) *indirim çarpanı* uygulamak suretiyle, söz konusu hareketten sonra gelen bütün ödüllerin toplamına dayanan bir hareketi hesaplamaktır. Bu indirilmiş ödüllerin toplamına hareketin getirisi denir. Şekil-18.6'daki örneği düşünün. Ajan art arda üç kez sağa gitmeye karar verirse ve ilk adımdan sonra +10 ödül, ikinci adımdan sonra 0 ve son olarak üçüncü adımdan sonra -50 alırsa, γ =0.8 olacak şekilde bir indirim çarpanı kullandığımızı varsayarak, ilk hareketin getirisi $10+ \gamma \times 0 + \gamma^2 \times(-50) = -22$ olur. İndirim çarpanı 0'a yakın ise gelecekteki ödüller anlık ödüllerle kıyaslandığında çok fazla sayılmaz. Aksine, indirim çarpanı 1'e yakınsa uzak gelecekteki indirimler hemen hemen dolaysız ödüller kadar sayılacaktır. Tipik indirim çarpanları 0.9'dan 0.99'a kadar değişir. 0.99'luk bir indirim çarpanı ile 69 adım gelecekteki ödüller anlık ödüllerin yarısı kadar sayılırken, 0.95'lik bir indirim çarpanı ile 13 adım gelecekteki ödüller anlık ödüllerin kabaca yarısı kadar sayılır ($0.95^{13} \approx 0.5$ olduğundan dolayı). CartPole ortamında hareketler oldukça kısa vadeli etkilere sahiptir. Dolayısıyla 0.95'lik bir indirim çarpanı kullanmak makul görünmektedir.



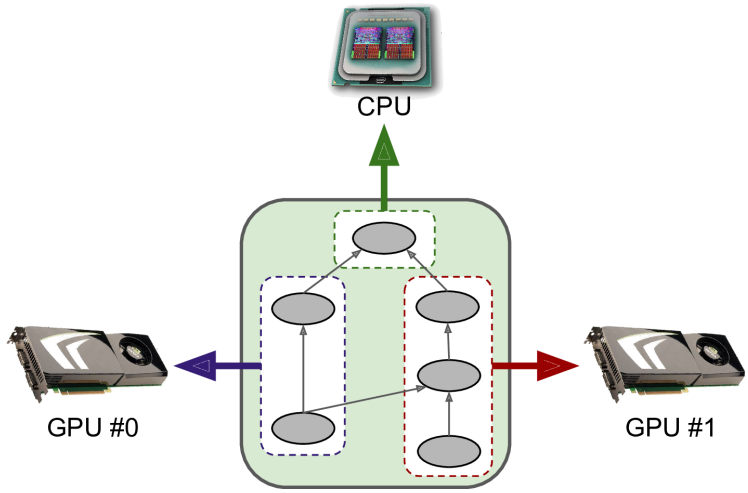
Şekil 18.6: *Bir hareketin getirisini hesaplamak: Gelecekteki indirilmiş ödüllerin toplamı*

Elbette ki iyi bir hareketi direğin çabucak düşmesine neden olan birkaç kötü hareket takip edebilir. Bu ise iyi hareketin düşük bir getiri elde etmesini sağlar (benzer şekilde iyi bir aktörün kötü bir filmde yıldızı parlayabilir). Bununla birlikte, oyunu yeterince oynarsak ortalamada iyi hareketler kötü hareketlerden daha yüksek bir getiri elde edecektir. Ortalamada, bir hareketin diğer muhtemel hareketlerle karşılaştırıldığında ne kadar iyi veya ne kadar kötü olduğunu kestirmek isteriz. Buna *hareket avantajı* denir. Bunun için pek çok epizod çalıştırmalı, bütün hareket getirilerini normalleştirmeliyiz (ortalamayı çıkarıp). Bundan sonra, makul bir şekilde negatif avantajlı hareketlerin kötü,

19.3 Hesaplamaları Hızlandırmak için GPU Kullanmak

Bölüm 11'de eğitimi oldukça hızlandırabilen farklı teknikler gördük: Ağırlıklara daha iyi ilk değer atama, Yığın Normalleştirme, daha karmaşık eniyileyiciler vb. Bütün bu teknikleri kullansanız bile büyük bir sinir ağını bir CPU'ya sahip bir makine de eğitmek günler hatta haftalar alabilir.

Bu kısımda modelleri hızlandırmada GPU'ların nasıl kullanılacağına bakacağız. Aynı zamanda bir CPU ve birden çok GPU'ya sahip birden çok makineye hesaplamaların dağıtılmasını inceleyeceğiz (Şekil-19.9'a bakınız). Şimdilik her şeyi tek bir makinede çalıştıracağız ve bölüm sonuna doğru birden çok sunucuya hesaplamaların nasıl dağıtılacağını tartışacağız.



Şekil 19.9: *Birden çok cihazda paralel olarak TensorFlow çizgesini çalıştırma*

GPU'lar sayesinde eğitim algoritmasının bitmesini günlerce ya da haftalarca beklemek yerine sadece birkaç dakika veya saat bekleyebilirsiniz. Bu sadece çok fazla zaman kazanma anlamına gelmez, aynı zamanda farklı modelleri deneyebilmeniz ve sıklıkla yeni veriler üzerinde modelinizi yeniden eğitebilmeniz anlamına gelir.

Genellikle tek bir makineye GPU kartları ekleyerek büyük bir performans artışı elde edebilirsiniz. Aslında çoğu durumda bu yeterlidir ve birden çok makine kullanmaya ihtiyacınız olmayacaktır. Örneğin, dağıtılmış bir kurulumdaki ağ iletişiminin gerektirdiği ekstra gecikme nedeniyle, tipik olarak bir sinir ağını, birden çok makine boyunca dağıtılmış 8 GPU yerine tek bir makinede dört GPU kullanarak aynı hızda eğitilebilirsiniz. Benzer şekilde, tek bir güçlü GPU kullanmak, genellikle, birden çok yavaş GPU kullanmaya tercih edilir

İlk adım olarak bir GPU kullanımıyla başlayalım. Bunun için iki seçenek var: Kendi GPU ya da GPU'larınızı satın almak ya da bulut üzerinde GPU ile donatılmış sanal makineler kullanmak. Şimdi ilk seçenekle başlayalım.

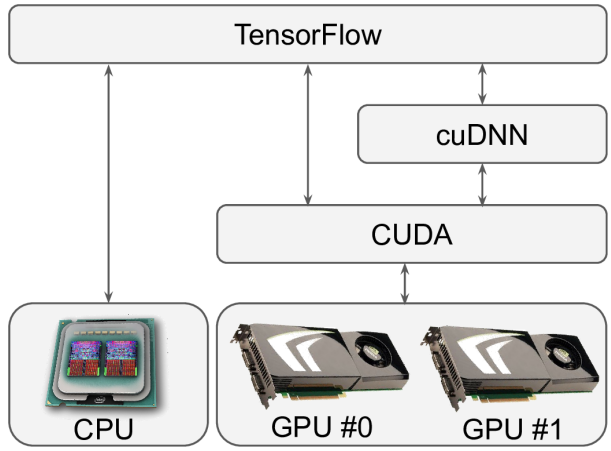
19.3.1 Kendi GPU'nuzu Alma

Eğer bir GPU kartı almayı seçerseniz doğru seçimi yapmak biraz zaman alacaktır. Tim Dettmers'ın seçim yapmanıza yardımcı olacak ve düzenli olarak güncellediği harika bir internet günlüğü bulunmaktadır (*https://homl.info/66*): Dikkatlice okumanızı tavsiye ederim. Bu kitabın yazıldığı sırada TensorFlow sadece CUDA Compute Capability 3.5 ve daha büyük Nvidia kartlara (*https://homl.info/cudagpus*) destek vermektedir (elbette aynı zamanda Google'ın TPU'larına) ama ileride diğer üreticileri de destekleyebilir. Her ne kadar TPU'lar şimdilik sadece GCP üzerinden ulaşılabilir olsalar da yakın gelecekte TPU benzeri kartlar satışa çıkabilir ve TensorFlow bu kartları destekleyebilir. Kısacası, bu noktada hangi cihazların desteklendiğini görmek için TensorFlow'un dokümantasyonunu (*https://tensorflow.org/install*) kontrol ettiğinizden emin olun.

Eğer bir Nvidia kartı seçerseniz Nvidia donanım sürücülerini ve bazı diğer Nvidia kütüphanelerini[10] kurmalısınız. Bunlar geliştiricilere CUDA destekli GPU'larda her türlü hesaplamayı (sadece grafik hızlandırma değil) yapma imkânı sağlayan *Compute Unified Device Architecture kütüphanesi* (CUDA) ve derin sinir ağlarındaki işlemleri için GPU hızlandırmaları sağlayan *CUDA Deep Neural Network* kütüphanesidir (cuDNN). cuDNN aktivasyon katmanları, normalizasyon, ileriye ve geriye evrişim ve biriktirme gibi çok sık kullanılan derin sinir ağı hesaplamaları için eniyilenmiş gerçekleştirmeler sağlar (Bölüm 14'e bakınız). Nvidia'nın Deep Learning SDK'sının bir parçasıdır (indirmek için Nvidia geliştirici hesabı oluşturmanız gerekecektir). TensorFlow, CUDA ve

[10] Güncel kurulum adımları ve detaylar için lütfen dokümantasyonu sıkça kontrol edin.

cuDNN kütüphanelerini GPU'ları kontrol etmek ve hesaplamaları hızlandırmak için kullanır (Şekil-19.10'a bakınız).



Şekil 19.10: *TensorFlow, GPU'ları kontrol etmek ve DNN'leri hızlandırmak için CUDA ve cuDNN kullanır*

GPU kartlarını (veya kartını) ve gerekli tüm sürücüleri ve kütüphaneleri yükledikten sonra, CUDA'nın doğru şekilde kurulup kurulmadığını kontrol etmek için `nvidia-smi` komutunu kullanabilirsiniz. Mevcut GPU kartlarını ve her kartta çalışan işlemleri listeler:

```
$ nvidia-{}smi
Sun Jun  2 10:05:22 2019
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.67 Driver Version: 410.79 CUDA Version: 10.0 |
|-------------------------------+----------------------+----------------------+
| GPU Name Persistence-M| Bus-Id Disp.A | Volatile Uncorr. ECC |
| Fan Temp Perf Pwr:Usage/Cap| Memory-Usage | GPU-Util Compute M. |
|===============================+======================+======================|
| 0 Tesla T4 Off | 00000000:00:04.0 Off | 0 |
| N/A 61C P8 17W / 70W | 0MiB / 15079MiB | 0% Default |
+-------------------------------+----------------------+----------------------+
+-----------------------------------------------------------------------------+
| Processes: GPU Memory |
| GPU PID Type Process name Usage |
|=============================================================================|
| No running processes found |
+-----------------------------------------------------------------------------+
```

Bu kitabın yazıldığı zaman için TensorFlow'un GPU versiyonunu da (örneğin `tensorflow-gpu` kütüphanesi) kurmaya ihtiyacınız vardı. Ancak sadece CPU'lu ve GPU'lu makinelere kurulumu birleştiren bir çalışma devam ediyordu. Dolayısıyla lütfen kurulum dokümantasyonunu inceleyerek hangi kütüphaneyi kurmanız gerektiğini kontrol edin. Her durumda tüm gerekli kütüphanelerin doğru şekilde kurulumu biraz zahmetli ve zaman alıcı olduğundan (eğer doğru versiyonlarını kurmamanız hâlinde içinden çıkılamaz hâle gelecektir) TensorFlow ihtiyacınız olan her şeyin içerisinde olduğu bir Docker görüntüsü sağlamaktadır. Ancak Docker konteyneri içerisinden GPU'ya erişmek için hâlâ ana makinenizde Nvidia donanım sürücülerinin kurulu olmasına ihtiyacınız olacaktır.

TensorFlow'un GPU'ları gerçekten gördüğünü kontrol etmek için aşağıdaki testleri çalıştırın:

```
>>> import tensorflow as tf
>>> tf.test.is_gpu_available()
True
>>> tf.test.gpu_device_name()
'/device:GPU:0'
>>> tf.config.experimental.list_physical_devices(device_type='GPU')
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

`is_gpu_available()` fonksiyonu en azından bir GPU'nun mevcut olup olmadığını kontrol eder. `gpu_device_name()` fonksiyonu ilk GPU'nun adını döndürür: Varsayılan olarak işlemler bu GPU üzerinde çalışacaktır. `list_physical_devices()` fonksiyonu tüm mevcut GPU cihazlarının listesini (bu örnekte bir tane var) döndürür.[11]

Şimdi eğer zamanınızı ve paranızı kendi GPU'nuza yatırmayı istemezseniz ne yapabileceğinize bakalım. Bulutta GPU'lu sanal makine kullanırsınız!

19.3.2 GPU'lu Sanal Makine Kullanımı

Tüm büyük bulut platformları önceden tüm donanım sürücüleri konfigüre edilmiş ve ihtiyacınız olan kütüphaneleri (TensorFlow gibi) barındıran GPU'lu sanal makineler sunmaktadır. Google Cloud Platform dünya çapında ve bölge bazında GPU kotaları uygular: Google'dan izin almadan binlerce GPU'lu

[11] Bu bölümdeki kod örneklerinin çoğu deneysel API'ları kullanmaktadır. İlerideki versiyonlarda temel API içine taşınma ihtimali bulunmaktadır. Sonuç olarak eğer deneysel bir metot hata verirse `experimental` kelimesi iç aktarmadan silip tekrar denerseniz muhtemelen çalışacaktır. Eğer bu da olmazsa API değişmiş olabilir. Her zaman doğru kodun olmasını sağlayacağımdan lütfen Jupyter not defterini kontrol edin.

Dizin