 olimpikbarbell@gmail.com

```
...     epoch_train_acc = []
...     epoch_valid_acc = []
...
...     for e in range(num_epochs):
...         # minibatchler üzerinde yinele
...         minibatch_gen = minibatch_generator(
...             X_train, y_train, minibatch_size)
...         for X_train_mini, y_train_mini in minibatch_gen:
...             #### Çıktıları hesapla ####
...             a_h, a_out = model.forward(X_train_mini)

...             #### Gradyanları hesaplayın ####
...             d_loss d_w_out, d_loss d_b_out, \
...             d_loss d_w_h, d_loss d_b_h = \
...                 model.backward(X_train_mini, a_h, a_out,
...                                y_train_mini)
...
...             #### Güncelleme ağırlıkları ####
...             model.weight_h -= learning_rate * d_loss d_w_h
...             model.bias_h -= learning_rate * d_loss d_b_h
...             model.weight_out -= learning_rate * d_loss d_w_out
...             model.bias_out -= learning_rate * d_loss d_b_out
...
...         #### Epoch Günlüğü ####
...         train_mse, train_acc = compute_mse_and_acc(
...             model, X_train, y_train
...         )
...         valid_mse, valid_acc = compute_mse_and_acc(
...             model, X_valid, y_valid
...         )
...         train_acc, valid_acc = train_acc*100, valid_acc*100
...         epoch_train_acc.append(train_acc)
...         epoch_valid_acc.append(valid_acc)
...         epoch_loss.append(train_mse)
...         print(f'Epoch: {e+1:03d}/{num_epochs:03d} '
...               f'| Train MSE: {train_mse:.2f} '
...               f'| Tren Acc: {train_acc:.2f}% '
...               f'| Geçerli Acc: {valid_acc:.2f}%')
...
...     epoch_loss, epoch_train_acc, epoch_valid_acc döndür
```

Yüksek seviyede, train() fonksiyonu birden fazla epok üzerinde yineleme yapar ve her epokta, stokastik gradyan inişi eğitimi için mini gruplar halinde tüm eğitim seti üzerinde yineleme yapmak üzere önceden tanımlanmış minibatch_generator() fonksiyonunu kullanır. Mini parti oluşturucu for döngüsü içinde, .forward() yöntemi aracılığıyla modelden a_h ve a_out çıktılarını elde ediyoruz. Ardından, modelin .backward() yöntemi aracılığıyla kayıp gradyanlarını hesaplarız - teori daha sonraki bir bölümde açıklanacaktır. Kayıp gradyanlarını kullanarak, öğrenme oranıyla çarpılan negatif gradyanı ekleyerek ağırlıkları güncelleriz. Bu, daha önce Adaline için tartıştığımız kavramla aynıdır. Örneğin, gizli katmanın model ağırlıklarını güncellemek için aşağıdaki satırı tanımladık:

```
model.weight_h -= learning_rate * d_loss d_w_h
```

Tek bir ağırlık için, w_j , bu aşağıdaki kısmi türev tabanlı güncellemeye karşılık gelir:

$$w_j := w_j - \eta \frac{\partial L}{\partial w_j}$$

Son olarak, önceki kodun son kısmı, eğitim ilerlemesini izlemek için eğitim ve test setleri üzerindeki kayıpları ve tahmin doğruluklarını hesaplar.

Şimdi modelimizi 50 epok boyunca eğitmek için bu işlevi çalıştıralım; bu işlemin tamamlanması birkaç dakika sürebilir:

```
>> np.random.seed(123) # eğitim setinin karıştırılması için
>> epoch_loss, epoch_train_acc, epoch_valid_acc = train(
...         model, X_train, y_train, X_valid, y_valid,
...         num_epochs=50, learning_rate=0.1)
```

Eğitim sırasında aşağıdaki çıktıyı görmeliyiz:

```
Epoch: 001/050 | Train MSE: 0.05 | Train Acc: 76.17% | Valid Acc: 76.02%
Epoch: 002/050 | Train MSE: 0.03 | Train Acc: 85.46% | Valid Acc: 84.94%
Epoch: 003/050 | Train MSE: 0.02 | Tren Acc: 87.89% | Valid Acc: 87.64%
Epoch: 004/050 | Tren MSE: 0.02 | Tren Acc: 89.36% | Valid Acc: 89.38%
Epoch: 005/050 | Tren MSE: 0.02 | Tren Acc: 90.21% | Valid A c c : 90.16%
...
Dönem: 048/050 | Tren MSE: 0.01 | Tren Acc: 95.57% | Geçerli Acc: 94.58%
Dönem: 049/050 | Tren MSE: 0.01 | Tren Acc: 95.55% | Geçerli Acc: 94.54%
Dönem: 050/050 | Tren MSE: 0 . 01 | Tren Acc: 9 5 . 59% | Geçerli A c c :
9 4 . 74%
```

Tüm bu çıktıları yazdırmamızın nedeni, NN eğitiminde, eğitim ve doğrulama doğruluğunu karşılaştırmanın gerçekten yararlı olmasıdır. Bu, mimari ve hiperparametreler göz önüne alındığında ağ modelinin iyi performans gösterip göstermediğini değerlendirmemize yardımcı olur. Örneğin, düşük bir eğitim ve doğrulama doğruluğu gözlemlersek, muhtemelen eğitim veri kümesiyle ilgili bir sorun vardır veya hiperparametrelerin ayarları ideal değildir.

Genel olarak, (derin) NN'leri eğitmek şimdiye kadar tartıştığımız diğer modellere kıyasla nispeten pahalıdır. Bu nedenle, belirli durumlarda erken durdurmak ve farklı hiper çap ayarlarıyla baştan başlamak istiyoruz. Öte yandan, eğitim verilerine giderek daha fazla uyum sağlama eğiliminde olduğunu tespit edersek (eğitim ve doğrulama veri kümesi performansı arasında artan bir boşluk ile

olimpikbarbell@gmail.com

fark edilir), eğitimi erken durdurmak da isteyebiliriz.

Bir sonraki alt bölümde, NN modelimizin performansını daha ayrıntılı olarak tartışacağız.

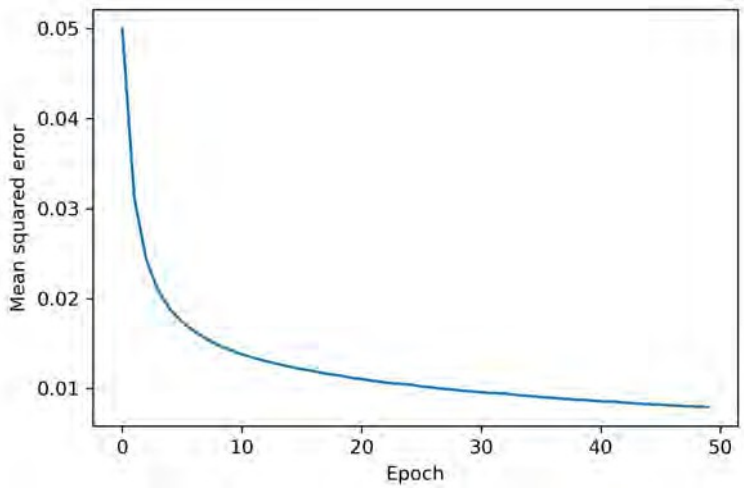
Sinir ağı performansının değerlendirilmesi

NN'lerin eğitim prosedürü olan geriye yayılımı bir sonraki bölümde daha ayrıntılı olarak tartışmadan önce, bir önceki alt bölümde eğittiğimiz modelin performansına bakalım.

`train()` içinde, Matplotlib kullanarak sonuçları görselleştirebilmemiz için her epok için eğitim kaybını ve eğitim ve doğrulama doğruluğunu topladık. Önce eğitim MSE kaybına bakalım:

```
>> plt.plot(range(len(epoch_loss)), epoch_loss)
>> plt.ylabel('Ortalama karesel hata')
>> plt.xlabel('Epoch')
>> plt.show()
```

Önceki kod, *Şekil 11.7*'de gösterildiği gibi 50 epok boyunca kaybı çizer:



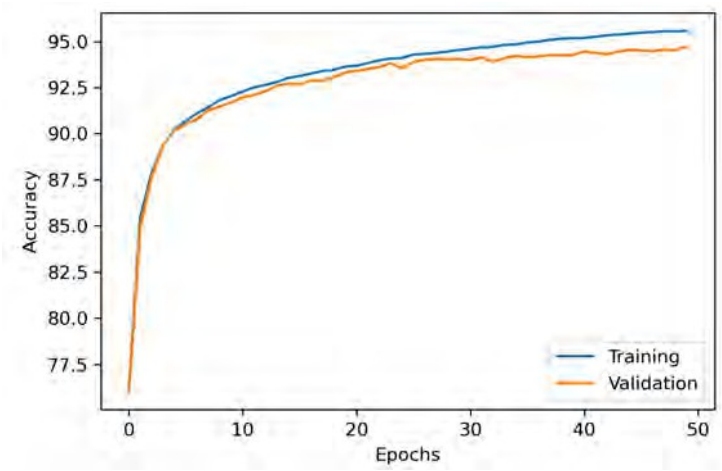
Şekil 11.7: Eğitim epoklarının sayısına göre MSE grafiği

Gördüğümüz gibi, kayıp ilk 10 epok boyunca önemli ölçüde azalmış ve son 10 epokta yavaşça yakınsıyor gibi görünmektedir. Bununla birlikte, epok 40 ile epok 50 arasındaki küçük eğim, kaybın ek epoklar boyunca eğitimle daha da azalacağını göstermektedir.

Daha sonra, eğitim ve doğrulama doğruluğuna bir göz atalım:

```
>> plt.plot(range(len(epoch_train_acc)), epoch_train_acc,
...              label='Eğitim')
>> plt.plot(range(len(epoch_valid_acc)), epoch_valid_acc,
...              label='Validation')
>> plt.ylabel('Doğruluk')
>> plt.xlabel('Çağlar')
>> plt.legend(loc='sağ alt')
>> plt.show()
```

Önceki kod örnekleri, bu doğruluk değerlerini 50 eğitim epoku üzerinden çizmektedir. *Şekil 11.8*:



Şekil 11.8: Eğitim epoklarının sayısına göre sınıflandırma doğruluğu

Grafik, daha fazla epok için eğitim yaptıkça eğitim ve doğrulama doğruluğu arasındaki farkın arttığını ortaya koymaktadır. Yaklaşık 25. epokta, eğitim ve doğrulama doğruluk değerleri neredeyse eşittir ve daha sonra ağ, eğitim verilerine biraz fazla uyum sağlamaya başlar.

Aşırı uyumun azaltılması

Aşırı uyumun etkisini azaltmanın *bir* yolu, *Bölüm 3, Scikit-Learn Kullanarak Makine Öğrenimi Sınıflandırıcıları Turu*'nda tanıttığımız L2 düzenlileştirme yoluyla düzenlileştirme gücünü artırmaktır. NN'lerde aşırı uyumun üstesinden gelmek için bir başka yararlı teknik de *Bölüm 14, Derin Evrişimli Sinir Ağları ile Görüntüleri Sınıflandırma'da* ele alınacak olan dropout'tur.

Son olarak, test veri kümesi üzerindeki tahmin başarısını hesaplayarak modelin genelleme performansını değerlendirelim:

```
>> test_mse, test_acc = compute_mse_and_acc(model, X_test, y_test)
>> print(f'Test doğruluğu: {test_acc*100:.2f}%')
Test doğruluğu: %94,51
```

Test doğruluğunun, son alt bölümde eğitim sırasında bildirdiğimiz son epoka (%94,74) karşılık gelen doğrulama seti doğruluğuna çok yakın olduğunu görebiliriz. Dahası, ilgili eğitim doğruluğu %95,59 ile yalnızca minimum düzeyde daha yüksektir ve modelimizin eğitim verilerine yalnızca biraz fazla uyum sağladığını teyit etmektedir.

Modele daha fazla ince ayar yapmak için gizli birimlerin sayısını, öğrenme oranını değiştirebilir veya yıllar içinde geliştirilen ancak bu kitabın kapsamı dışında kalan çeşitli diğer hileleri kullanabiliriz. *Derin Evrişimli Sinir Ağları ile Görüntülerin Sınıflandırılması* başlıklı *14. Bölümde,* görüntü veri kümelerinde iyi performans göstermesiyle bilinen farklı bir NN mimarisi hakkında bilgi edineceksiniz.

Ayrıca, bu bölümde uyarlanabilir öğrenme oranları, daha sofistike SGD tabanlı optimizasyon algoritmaları, toplu normalleştirme ve bırakma gibi performans artırıcı ek hileler tanıtılacaktır.

Aşağıdaki bölümlerin kapsamı dışında kalan diğer yaygın hileler şunlardır:

- Artık NN'lerin ana katkısı olan atlama bağlantılarının eklenmesi (*Deep residual learn- ing for image recognition* by *K. He*, *X. Zhang*, *S. Ren*, and *J. Sun, Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition,* pp. 770-778, 2016)
- Eğitim sırasında öğrenme oranını değiştiren öğrenme oranı zamanlayıcılarının kullanılması (*Sinir ağlarının eğitimi için döngüsel öğrenme oranları*, *L.N. Smith*, *2017 IEEE Winter Conference on Applications of Computer Vision (WACV),* s. 464-472, 2017)
- Popüler *In-* ception v3 mimarisinde yapıldığı gibi ağlardaki daha önceki katmanlara kayıp fonksiyonlarının eklenmesi (*Rethinking the Inception architecture for computer vision*, *C. Szegedy*, *V. Vanhoucke, S. Ioffe, J. Shlens*, and *Z. Wojna, Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2818-2826, 2016)

Son olarak, test setinden ilk 25 yanlış sınıflandırılmış örneği çıkarıp çizerek MLP'mizin zorlandığı bazı görüntülere bir göz atalım:

```
>> X_test_subset = X_test[:1000, :]
>> y_test_subset = y_test[:1000]
>> _, probas = model.forward(X_test_subset)
>>> test_pred = np.argmax(probas, axis=1)
>>> yanlış sınıflandırılmış_görüntüler = \
...         X_test_subset[y_test_subset != test_pred][:25]
>>> misclassified_labels = test_pred[y_test_subset != test_pred][:25]
>> correct_labels = y_test_subset[y_test_subset != test_pred][:25]

>> fig, ax = plt.subplots(nrows=5, ncols=5,
...                        sharex=True, sharey=True,
...                        figsize=(8, 8))
>> ax = ax.flatten()
>> for i  in range(25):
...       img = misclassified_images[i].reshape(28, 28)
...       ax[i].imshow(img, cmap='Greys', interpolation='nearest')
...       ax[i].set_title(f'{i+1}) '
...                       f'True: {correct_labels[i]}\n'
...                       f' Tahmin edilen: {misclassified_labels[i]}')

>>  ax[0].set_xticks([])
>>  ax[0].set_yticks([])
>>  plt.tight_layout()
>> plt.show()
```

alinpikbarbell@gmail.com

Şimdi, alt başlıklardaki ilk sayının çizim indeksini, ikinci sayının gerçek sınıf etiketini (True) ve üçüncü sayının tahmin edilen sınıf etiketini (Predicted) temsil ettiği 5×5'lik bir alt grafik matrisi görmeliyiz:

Şekil 11.9: Modelin doğru sınıflandıramadığı el yazısı rakamlar

*Şekil 11.9'*da görebileceğimiz gibi, diğerlerinin yanı sıra, 19 ve 20 numaralı örneklerde olduğu gibi yatay bir çizgi içerdiklerinde ağ 7'leri zorlayıcı bulmaktadır. Bu bölümde 7 rakamının farklı eğitim örneklerini çizdiğimiz daha önceki bir şekle bakarak, yatay çizgi içeren el yazısı 7 rakamının veri setimizde yeterince temsil edilmediğini ve genellikle yanlış sınıflandırıldığını varsayabiliriz.

Yapay sinir ağının eğitilmesi

Artık bir NN'yi çalışırken gördüğümüze ve kodu inceleyerek nasıl çalıştığına dair temel bir anlayış kazandığımıza göre, model parametrelerini öğrenmek için uyguladığımız kayıp hesaplama ve geriye yayılma algoritması gibi bazı kavramları biraz daha derinlemesine inceleyelim.

Kayıp fonksiyonunun hesaplanması

Daha önce de belirtildiği gibi, çok katmanlı NN'yi eğitmek için MSE kaybı (Adaline'de olduğu gibi) kullandık çünkü bu, gradyanların türetilmesini takip etmeyi biraz daha kolaylaştırıyor. Daha sonraki bölümlerde, NN sınıflandırıcılarını eğitmek için daha yaygın bir seçim olan çok kategorili çapraz entropi kaybı (ikili lojistik regresyon kaybının bir genellemesi) gibi diğer kayıp fonksiyonlarını tartışacağız.

Önceki bölümde, çok sınıflı sınıflandırma için, tek-sıcak kodlama temsilinde $t \times 1$ boyutlu hedef vektörle karşılaştırmamız gereken t elemanlı bir çıktı vektörü döndüren bir MLP uyguladık. Bu MLP'yi kullanarak sınıf etiketi 2 olan bir girdi görüntüsünün sınıf etiketini tahmin edersek, üçüncü katmanın ve hedefin aktivasyonu aşağıdaki gibi görünebilir:

$$a^{(out)} = \begin{bmatrix} 0.1 \\ 0.9 \\ \vdots \\ 0.3 \end{bmatrix}, \; y = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

Bu nedenle, MSE kaybımız, veri kümesindeki n örnek veya mini yığın üzerinde ortalama almanın yanı sıra ağımızdaki t aktivasyon birimi üzerinde toplam veya ortalama almak zorundadır:

$$L(\boldsymbol{W}, \boldsymbol{W}, \boldsymbol{b}) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{t} \sum_{j=1}^{t} \left(y_j^{[i]} - a_j^{(out)[i]} \right)^2$$

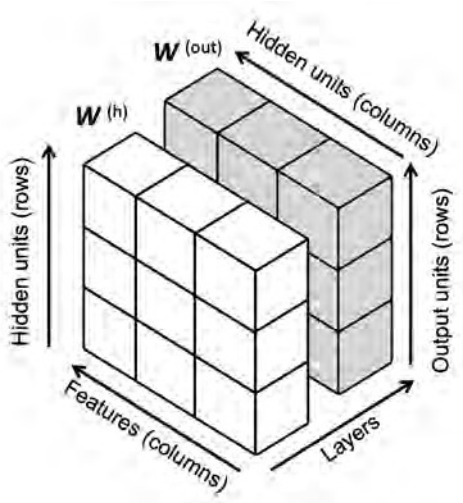
Burada da üst simge $[i]$, eğitim veri setimizdeki belirli bir örneğin indeksidir.

Amacımızın $L(\boldsymbol{W})$ kayıp fonksiyonunu minimize etmek olduğunu unutmayın; bu nedenle, ağdaki her katman için her ağırlığa göre $\boldsymbol{W}$ parametrelerinin kısmi türevini hesaplamamız gerekir:

$$\frac{\partial}{\partial \partial_{ij}^{(l)}} L(\boldsymbol{W}, \boldsymbol{W}, \boldsymbol{b})$$

Bir sonraki bölümde, kayıp fonksiyonunu en aza indirmek için bu kısmi türevleri hesaplamamızı sağlayan geriye yayılma algoritmasından bahsedeceğiz.

W'nin birden fazla matristen oluştuğunu unutmayın. Tek gizli katmanlı bir MLP'de, girişi gizli katmana bağlayan $W^{(h)}$ ve gizli katmanı çıkış katmanına bağlayan $W^{(out)}$ ağırlık matrisine sahibiz. Üç boyutlu $\boldsymbol{W}$ tensörünün görselleştirmesi *Şekil 11.10'da* verilmiştir:



Şekil 11.10: Üç boyutlu bir tensörün görselleştirilmesi

Bu basitleştirilmiş şekilde, hem $W^{(h)}$ hem de $W^{(out)}$ aynı sayıda satır ve sütuna sahip gibi görünebilir, ancak bir MLP'yi aynı sayıda gizli birim, çıktı birimi ve giriş özelliği ile başlatmadığımız sürece durum genellikle böyle değildir.

Bu size kafa karıştırıcı geliyorsa, $W^{(h)}$ ve $W^{(out)}$ boyutlarını geriye yayılma algoritması bağlamında daha ayrıntılı olarak tartışacağımız bir sonraki bölüm için bizi izlemeye devam edin. Ayrıca, farklı matrislerin ve vektör dönüşümlerinin boyutluluğu hakkında yararlı yorumlarla açıklanan NeuralNetMLP kodunu tekrar okumanız önerilir.

Geriye yayılım anlayışınızı geliştirme

Geriye yayılım, sinir ağı topluluğuna 30 yıldan daha uzun bir süre önce tanıtılmış olmasına rağmen (*Learning representations by backpropagating errors*, *D.E. Rumelhart*, *G.E. Hinton,* and *R.J. Williams, Nature*, 323: 6088, pages 533-536, 1986), yapay NN'leri çok verimli bir şekilde eğitmek için en yaygın kullanılan algoritmalardan biri olmaya devam etmektedir. Geriye yayılımın tarihçesine ilişkin ek referanslarla ilgileniyorsanız, Juergen Schmidhuber, http://people.idsia.ch/~juergen/who-invented-backpropagation. html adresinde çevrimiçi olarak bulabileceğiniz Geriye Yayılımı *Kim İcat* Etti*?*

Bu bölüm, daha fazla matematiksel ayrıntıya girmeden önce bu büyüleyici algoritmanın nasıl çalıştığına dair hem kısa ve net bir özet hem de daha büyük bir resim sunacaktır. Özünde, geriye yayılımı çok katmanlı NN'lerde karmaşık, konveks olmayan bir kayıp fonksiyonunun kısmi türevlerini hesaplamak için hesaplama açısından çok verimli bir yaklaşım olarak düşünebiliriz. Burada amacımız, bu türevleri kullanarak böyle bir çok katmanlı yapay NN'yi parametrelendirmek için ağırlık katsayılarını öğrenmektir. NN'lerin parametrelendirilmesindeki zorluk, tipik olarak yüksek boyutlu bir özellik uzayında çok sayıda model parametresiyle uğraşıyor olmamızdır. Önceki bölümlerde gördüğümüz Adaline veya lojistik regresyon gibi tek katmanlı NN'lerin kayıp fonksiyonlarının aksine, bir NN kayıp fonksiyonunun hata yüzeyi parametrelere göre dışbükey veya pürüzsüz değildir. Bu yüksek boyutlu kayıp yüzeyinde, kayıp fonksiyonunun global minimumunu bulmak için üstesinden gelmemiz gereken birçok tümsek (yerel minimum) vardır.

Zincir kuralı kavramını kalkülüse giriş derslerinizden hatırlayabilirsiniz. Zincir kuralı, $f(g(x))$ gibi karmaşık, iç içe geçmiş bir fonksiyonun türevini hesaplamak için aşağıdaki gibi bir yaklaşımdır:

$$\frac{d}{dx}[f(g(x))] = \frac{df}{dg} \cdot \frac{dg}{dx}$$

Benzer şekilde, zincir kuralını keyfi uzunlukta bir fonksiyon bileşimi için de kullanabiliriz. Örneğin, elimizde $f(x)$, $g(x)$, $h(x)$, $u(x)$ ve $v(x)$ *olmak üzere* beş farklı fonksiyon olduğunu varsayalım ve F *fonksiyon bileşimi olsun:* $F(x) = f(g(h(u(v(x)))))$. Zincir kuralını uygulayarak, bu fonksiyonun türevini aşağıdaki gibi hesaplayabiliriz:

$$\frac{dF}{dx} = \frac{d}{dx}F(x) = \frac{d}{dx}f(g(h(u(v(x))))) = \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{du} \cdot \frac{du}{dv} \cdot \frac{dv}{dx}$$

Bilgisayar cebiri bağlamında, bu tür problemleri çok verimli bir şekilde çözmek için **otomatik türevlendirme olarak** bilinen bir dizi teknik geliştirilmiştir. Makine öğrenimi uygulamalarında otomatik türev hakkında daha fazla bilgi edinmek istiyorsanız, A.G. Baydin ve B.A. Pearlmutter'in makalesini okuyun, *Automatic Differentiation of Algorithms for Machine Learning,* arXiv preprint arXiv:1404.7456, 2014, arXiv'de http://arxiv.org/pdf/1404.7456.pdf adresinden ücretsiz olarak erişilebilir.

Otomatik farklılaştırma, ileri ve geri mod olmak üzere iki modla birlikte gelir; geriye yayılma basitçe ters mod otomatik farklılaştırmanın özel bir durumudur. Kilit nokta, zincir kuralını ileri modda uygulamanın oldukça pahalı olabileceğidir, çünkü her katman için çıktıyı elde etmek için sonunda bir vektörle çarpacağımız büyük matrisleri (Jacobianlar) çarpmamız gerekecektir.

Ters modun püf noktası, zincir kuralını sağdan sola doğru kat etmemizdir. Bir matrisi bir vektörle çarparız, bu da bir sonraki matrisle çarpılan başka bir vektör verir ve bu böyle devam eder. Matris-vektör çarpımı hesaplama açısından matris-matris çarpımından çok daha ucuzdur, bu nedenle geri yayılım NN eğitiminde kullanılan en popüler algoritmalardan biridir.

Temel kalkülüs tazeleme

Geriye yayılımı tam olarak anlamak için, bu kitabın kapsamı dışında olan diferansiyel hesaptan bazı kavramları ödünç almamız gerekir. Ancak, bu bağlamda faydalı bulabileceğiniz en temel kavramların gözden geçirildiği bir bölüme başvurabilirsiniz. Bu bölümde fonksiyon türevleri, kısmi türevler, gradyanlar ve Jacobian tartışılmaktadır. Bu metne https://sebastianraschka.com/pdf/books/dlb/appendix_d_calculus.pdf adresinden ücretsiz olarak erişilebilir. Kalkülü bilmiyorsanız veya kısa bir tazelemeye ihtiyacınız varsa, bir sonraki bölümü okumadan önce bu metni ek bir destekleyici kaynak olarak okumayı düşünün.

Geriye yayılım yoluyla sinir ağlarının eğitimi

Bu bölümde, bir NN'deki ağırlıkları çok verimli bir şekilde nasıl öğrenebileceğinizi anlamak için geriye yayılımın matematiğinden geçeceğiz. Matematiksel ifadelerle ne kadar rahat olduğunuza bağlı olarak, aşağıdaki denklemler ilk başta nispeten karmaşık görünebilir.

Bir önceki bölümde, kaybı son katmanın aktivasyonu ile hedef sınıf etiketi arasındaki fark olarak nasıl hesaplayacağımızı gördük. Şimdi, NeuralNetMLP() sınıfının .backward() yönteminde uyguladığımız geri yayılım algoritmasının MLP modelimizdeki ağırlıkları matematiksel bir perspektiften güncellemek için nasıl çalıştığını göreceğiz. Bu bölümün başından beri hatırladığımız gibi, aşağıdaki gibi formüle ettiğimiz çıktı katmanının aktivasyonunu elde etmek için öncelikle ileri yayılım uygulamamız gerekir:

$$Z^{(h)} = XX^{(in)})WW^{(h)T} + b^{(h)}$$ (gizli katmanın net girişi)

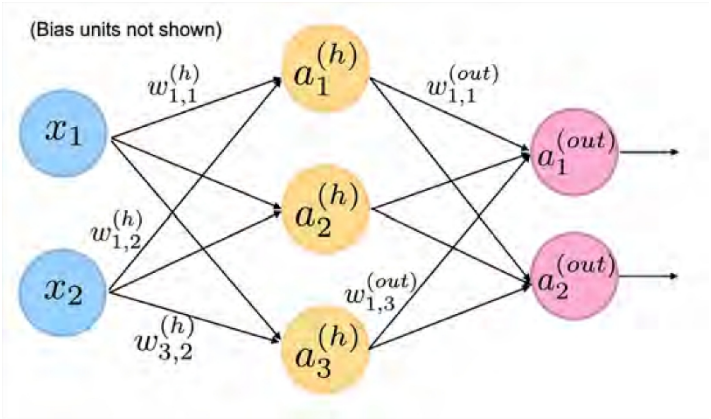
$$A^{(h)} = \sigma\sigma Z)^{(h)}$$ (gizli katmanın aktivasyonu)

$$Z^{(out)} = A^{(h)}\ WW(out)^{T} + b(out)$$ (çıktı katmanının net girdisi)

$$A^{(out)} = \sigma\sigma Z^{(out)})$$ (çıktı katmanının aktivasyonu)

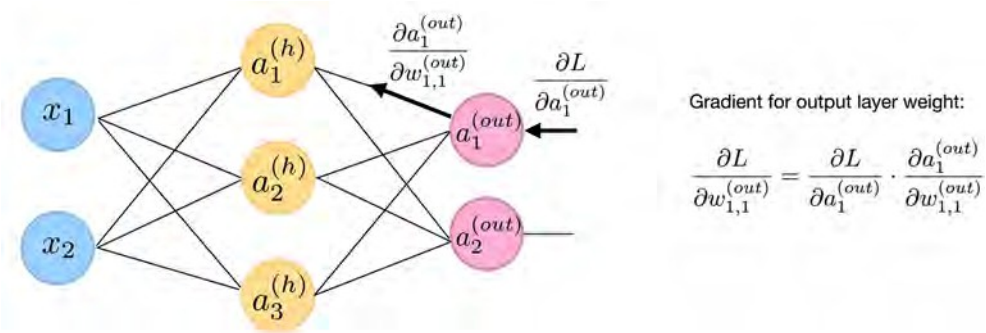
olimpikbarbell@gmail.com

Kısaca, iki giriş özelliği, üç gizli düğüm ve iki çıkış düğümü olan bir ağ için *Şekil 11.11*'deki oklarla gösterildiği gibi, giriş özelliklerini ağdaki bağlantılar aracılığıyla ileriye doğru yayarız:



Şekil 11.11: Bir NN'nin giriş özelliklerinin ileriye doğru yayılması

Geriye yayılımda hatayı sağdan sola doğru yayarız. Bunu, model ağırlıklarına (ve bias birimlerine) göre kaybın gradyanını hesaplamak için ileri geçişin hesaplanmasında zincir kuralının bir uygulaması olarak düşünebiliriz. Basit olması açısından, bu işlemi çıkış katmanının ağırlık matrisindeki ilk ağırlığı güncellemek için kullanılan kısmi türev için göstereceğiz. Geriye doğru ilerlettiğimiz hesaplama yolları aşağıdaki kalın oklarla vurgulanmıştır:



Şekil 11.12: Bir NN'nin hatasının geriye yayılması

Net z girdilerini açıkça dahil edersek, önceki şekilde gösterilen kısmi türev hesaplaması aşağıdaki gibi genişler:

$$\frac{\partial L}{\partial w_{1,1}^{(out)}} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial z_1^{(out)}} \cdot \frac{\partial z_1^{(out)}}{\partial w_{1,1}^{(out)}}$$

$w_{1,1}^{(out)}$'yi güncellemek için kullanılan bu kısmi türevi hesaplamak için, üç ayrı kısmi türev terimini hesaplayabilir ve sonuçları çarpabiliriz. Basitlik için, türevlerin ortalamasını almayacağız. mini partideki bireysel örnekler, bu nedenle $\frac{1}{n}\sum_{i=1}^{n}$ ortalama terimini aşağıdaki denklemlerden çıkarıyoruz.

Şunlarla başlayalım $\frac{\partial L}{\partial a_1^{(out)}}$ MSE kaybının kısmi türevi olan (karesel MSE kaybına basitleştiren) mini parti boyutunu ihmal edersek hata) ilk çıktı düğümünün tahmin edilen çıktı puanına göre:

$$\frac{\partial L}{\partial a_1^{(out)}} = \frac{\partial}{\partial a_1^{(out)}} (y_1 - a_1^{(out)})^2 = 2(a_1^{(out)} - y)$$

Bir sonraki terim, çıkış katmanında kullandığımız lojistik sigmoid aktivasyon fonksiyonunun türevidir:

$$\frac{\partial a_1^{(out)}}{\partial z_1^{(out)}} = \frac{\partial}{\partial z_1^{(out)}} \frac{1}{1 + e^{z_1^{(out)}}} = \ldots = \left(\frac{1}{1 + e^{z_1^{(out)}}}\right)\left(1 - \frac{1}{1 + e^{z_1^{(out)}}}\right)$$

$$= a_1^{(out)}(1 - a_1^{(out)})$$

Son olarak, net girdinin ağırlığa göre türevini hesaplarız:

$$\frac{\partial z_1^{(out)}}{\partial w_{1,1}^{(out)}} = \frac{\partial}{\partial w_{1,1}^{(out)}} a^{(h)} w_{1,1}^{(out)} + b_1^{(out)} = a_1^{(h)}$$

Tüm bunları bir araya getirdiğimizde şu sonuca ulaşıyoruz:

$$\frac{\partial L}{\partial w_{1,1}^{(out)}} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial z_1^{(out)}} \cdot \frac{\partial z_1^{(out)}}{\partial w_{1,1}^{(out)}} = 2(a_1^{(out)} - y) \cdot a_1^{(out)}(1 - a_1^{(out)}) \cdot a_1^{(h)}$$

Daha sonra bu değeri, η öğrenme oranıyla bilinen stokastik gradyan inişi güncellemesi yoluyla ağırlığı güncellemek için kullanırız:

$$w_{1,1}^{(out)} := w_{1,1}^{(out)} - \eta \frac{\partial L}{\partial w_{1,1}^{(out)}}$$

NeuralNetMLP() kod uygulamamızda, hesaplamayı şu şekilde gerçekleştirdik $\frac{\partial L}{\partial w_{1,1}^{(out)}}$ vektörize olarak .backward() metodunda aşağıdaki gibi form oluşturur:

```
# Bölüm 1: dLoss/dOutWeights
## = dLoss/dOutAct * dOutAct/dOutNet * dOutNet/dOutWeight
## burada DeltaOut = dLoss/dOutAct * dOutAct/dOutNet uygun yeniden kullanım
için

# girdi/çıktı dim: [n_examples, n_classes]
d_loss d_a_out = 2.*(a_out – y_onehot) / y.shape[0]

# girdi/çıktı dim: [n_examples, n_classes]
d_a_out d_z_out = a_out * (1. - a_out) # sigmoid türev

# çıktı dim: [n_examples, n_classes]
delta_out = d_loss d_a_out * d_a_out d_z_out # "delta (kural)
                                                # yer tutucu"

# çıktı ağırlıkları için gradyan #

[n_examples, n_hidden]
```

ilinpikbarbell@gmail.com

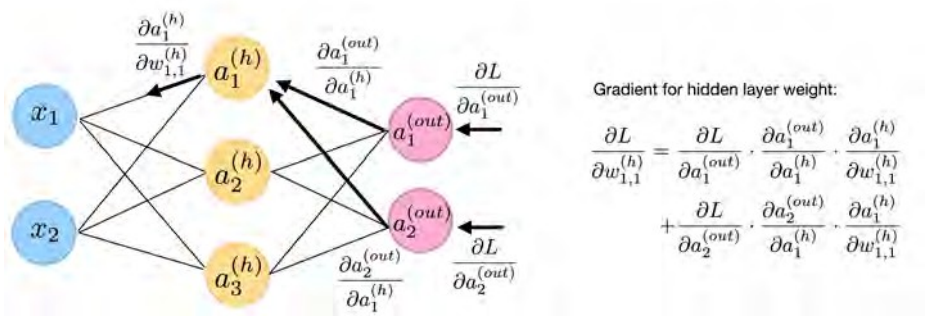
```
d_z_out dw_out = a_h

# girdi dim: [n_classes, n_examples] nokta [n_examples, n_hidden]
# çıktı d i m : [n_classes, n_hidden]
d_loss dw_out = np.dot(delta_out.T, d_z_out dw_out)
d_loss db_out = np.sum(delta_out, axis=0)
```

Yukarıdaki kod parçasında açıklandığı gibi, aşağıdaki "delta" yer tutucu değişkenini oluşturduk:

$$\delta_1^{(out)} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial z_1^{(out)}}$$

Bunun nedeni, $\delta^{(out)}$ terimlerinin gizli katman ağırlıklarının kısmi türevlerinin (veya gradyanlarının) hesaplanmasında da yer almasıdır; dolayısıyla, $\delta^{(out)}$'yi yeniden kullanabiliriz.

Gizli katman ağırlıklarından bahsetmişken, *Şekil 11.13* gizli katmanın ilk ağırlığına göre kaybın kısmi türevinin nasıl hesaplanacağını göstermektedir:



Şekil 11.13: Birinci gizli katman ağırlığına göre kaybın kısmi türevlerinin hesaplanması

$w_{1,1}^{(h)}$ ağırlığı her iki çıkış düğümüne de bağlı olduğundan, kalın oklarla vurgulanan iki yolu toplamak için *çok değişkenli* zincir kuralını kullanmamız gerektiğini vurgulamak önemlidir. Daha önce olduğu gibi, bunu net *z* girdilerini içerecek şekilde genişletebilir ve ardından tek tek terimleri çözebiliriz:

$$\frac{\partial L}{\partial w_{1,1}^{(h)}} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial z_1^{(out)}} \cdot \frac{\partial z_1^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial z_1^{(h)}} \cdot \frac{\partial z_1^{(h)}}{\partial w_{1,1}^{(h)}} + \frac{\partial L}{\partial a_2^{(out)}} \cdot \frac{\partial a_2^{(out)}}{\partial z_2^{(out)}} \cdot \frac{\partial z_2^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial z_1^{(h)}} \cdot \frac{\partial z_1^{(h)}}{\partial w_{1,1}^{(h)}}$$

Daha önce hesaplanan $\delta^{(out)}$ tekrar kullanılırsa, bu denklemin aşağıdaki gibi basitleştirilebileceğine dikkat edin:

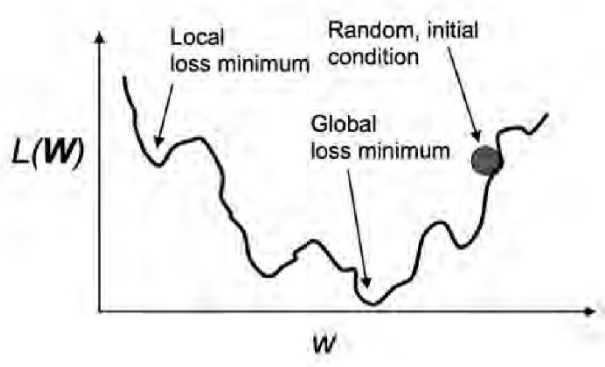
$$\frac{\partial L}{\partial w_{1,1}^{(h)}} = \delta_1^{(out)} \cdot \frac{\partial z_1^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial z_1^{(h)}} \cdot \frac{\partial z_1^{(h)}}{\partial w_{1,1}^{(h)}} + \delta_2^{(out)} \cdot \frac{\partial z_2^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial z_1^{(h)}} \cdot \frac{\partial z_1^{(h)}}{\partial w_{1,1}^{(h)}}$$

Daha önce yaptığımız gibi, önceki terimler tek tek nispeten kolay bir şekilde çözülebilir, çünkü yeni türevler söz konusu değildir. Örneğin, $\frac{\partial a_1^{(h)}}{\partial a_1^{(h)}} = 1$ sigmoid aktivasyonun türevidir, yani ise, $a_1^{(h)}(1 - a_1^{(h)})$ ve benzeri. Parçaları tek tek çözmeyi sizin için isteğe bağlı bir alıştırma olarak bırakacağız.

Sinir ağlarında yakınsama hakkında

Neden normal gradyan inişi kullanmadığımızı, bunun yerine daha önce el yazısı rakam sınıflandırması için NN'mizi eğitmek için mini toplu öğrenme kullandığımızı merak ediyor olabilirsiniz. Çevrimiçi öğrenmeyi uygulamak için kullandığımız SGD hakkındaki tartışmamızı hatırlayabilirsiniz. Çevrimiçi öğrenmede, ağırlık güncellemesini gerçekleştirmek için her seferinde tek bir eğitim örneğine ($k = 1$) dayalı olarak gradyanı hesaplarız. Bu stokastik bir yaklaşım olmasına rağmen, genellikle normal gradyan inişinden çok daha hızlı bir yakınsama ile çok doğru çözümlere yol açar. Mini toplu öğrenme, $1 < k < n$ olan n eğitim örneğinin k alt kümesine dayalı olarak gradyanı hesapladığımız özel bir SGD biçimidir. Mini toplu öğrenme, hesaplama verimliliğini artırmak için vektörleştirilmiş uygulamalarımızı kullanabilmemiz açısından çevrimiçi öğrenmeye göre bir avantaja sahiptir. Bununla birlikte, ağırlıkları normal gradyan inişine göre çok daha hızlı güncelleyebiliriz. Sezgisel olarak, mini toplu öğrenmeyi, tüm nüfusa sormak yerine (ki bu gerçek seçimi yürütmeye eşit olacaktır) nüfusun yalnızca temsili bir alt kümesine sorarak bir anketten bir başkanlık seçiminin seçmen katılımını tahmin etmek olarak düşünebilirsiniz.

Çok katmanlı NN'leri eğitmek Adaline, lojistik regresyon veya destek vektör makineleri gibi daha basit algoritmalara göre çok daha zordur. Çok katmanlı NN'lerde genellikle optimize etmemiz gereken yüzlerce, binlerce hatta milyarlarca ağırlık vardır. Ne yazık ki, çıktı fonksiyonu pürüzlü bir yüzeye sahiptir ve optimizasyon algoritması *Şekil 11.14*'te gösterildiği gibi kolayca yerel minimumlarda sıkışıp kalabilir:



Şekil 11.14: Optimizasyon algoritmaları yerel minimumlarda sıkışıp kalabilir

NN'mizin birçok boyutu olduğu için bu gösterimin son derece basitleştirilmiş olduğunu unutmayın; insan gözü için gerçek kayıp yüzeyini görselleştirmeyi imkansız hale getirir. Burada, x ekseninde yalnızca tek bir ağırlık için kayıp yüzeyini gösteriyoruz. Ancak ana mesaj, algoritmamızın yerel minimumlarda sıkışıp kalmasını istemediğimizdir. Öğrenme oranını artırarak, bu tür yerel minimumlardan daha kolay kaçabiliriz. Öte yandan, öğrenme oranı çok büyükse küresel optimumun aşılması olasılığını da artırırız. Ağırlıkları rastgele başlattığımız için, optimizasyon problemine tipik olarak umutsuzca yanlış olan bir çözümle başlarız.

 olimpikbarbell@gmail.com

Sinir ağı uygulaması hakkında son birkaç söz

Açık kaynaklı bir Python makine öğrenimi kütüphanesi kullanmak yerine neden sadece el yazısı rakamları sınıflandırabilen basit bir çok katmanlı yapay ağ uygulamak için tüm bu teoriye girdiğimizi merak ediyor olabilirsiniz. Aslında, sonraki bölümlerde açık kaynaklı PyTorch kütüphanesini (https://pytorch.org) kullanarak eğiteceğimiz daha karmaşık NN modellerini tanıtacağız.

Bu bölümdeki sıfırdan uygulama ilk başta biraz sıkıcı görünse de, geri yayılım ve NN eğitiminin arkasındaki temelleri anlamak için iyi bir alıştırmaydı. Algoritmaların temel düzeyde anlaşılması, makine öğrenimi tekniklerinin uygun ve başarılı bir şekilde uygulanması için çok önemlidir.

Artık ileri beslemeli NN'lerin nasıl çalıştığını öğrendiğinize göre, *Bölüm 12,* PyTorch *ile Sinir Ağı Eğitimini Paralelleştirme'*de göreceğimiz gibi, NN'leri daha verimli bir şekilde oluşturmamızı sağlayan PyTorch kullanarak daha karmaşık DNN'leri keşfetmeye hazırız.

İlk olarak Eylül 2016'da piyasaya sürülen PyTorch, **grafik işleme birimleri** (G**PU'lar**) kullanan çok boyutlu dizilerdeki hesaplamalar için matematiksel ifadeleri optimize etme yeteneği nedeniyle DNN'leri oluşturmak için kullanan ma- chine öğrenme araştırmacıları arasında çok popülerlik kazanmıştır.

Son olarak, scikit-learn'in https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html adresinde bulabileceğiniz temel bir MLP uygulaması olan MLPClassifier'ı da içerdiğini belirtmeliyiz. Bu uygulama temel MLP'leri eğitmek için harika ve çok kullanışlı olsa da, çok katmanlı NN'leri uygulamak ve eğitmek için PyTorch gibi özel derin öğrenme kütüphanelerini şiddetle tavsiye ediyoruz.

Özet

Bu bölümde, makine öğrenimi araştırmalarında şu anda en sıcak konu olan çok katmanlı yapay NN'lerin arkasındaki temel kavramları öğrendiniz. *Bölüm 2*, *Sınıflandırma için Basit Makine Öğrenimi Algoritmalarının Eğitimi'*nde, yolculuğumuza basit tek katmanlı NN yapıları ile başladık ve şimdi el yazısı rakam tanıma gibi karmaşık problemleri çözmek için güçlü bir NN mimarisine birden fazla nöron bağladık. Derin öğrenmede kullanılan birçok NN modelinin yapı taşlarından biri olan popüler backpropagation algoritmasının gizemini çözdük. Bu bölümde geri yayılım algoritmasını öğrendikten sonra, daha karmaşık DNN mimarilerini keşfetmek için iyi bir donanıma sahip olduk. Kalan bölümlerde, daha gelişmiş derin öğrenme kavramlarını ve çok katmanlı NN'leri daha verimli bir şekilde uygulamamızı ve eğitmemizi sağlayan açık kaynaklı bir kütüphane olan PyTorch'u ele alacağız.

Kitabımızın Discord alanına katılın

Benzer düşünen insanlarla tanışmak ve 2000'den fazla üyemizle birlikte öğrenmek için Discord topluluğumuza katılın:
https://packt.link/MLwPyTorch

6587191CA584E
olimpikbarbell@gmail.com

12

PyTorch ile Sinir Ağı Eğitimini Paralelleştirme

Bu bölümde, PyTorch'a odaklanmak için makine öğrenimi ve derin öğrenmenin matematiksel temellerinden geçeceğiz. PyTorch şu anda mevcut olan en popüler derin öğrenme kütüphanelerinden biridir ve **sinir ağlarını (NN'ler)** önceki NumPy uygulamalarımızdan çok daha verimli bir şekilde uygulamamızı sağlar. Bu bölümde PyTorch'u kullanmaya başlayacağız ve eğitim performansına nasıl önemli faydalar sağladığını göreceğiz.

Bu bölüm, makine öğrenimi ve derin öğrenme yolculuğumuzun bir sonraki aşamasına başlayacak ve aşağıdaki konuları inceleyeceğiz:

- PyTorch eğitim performansını nasıl artırır?
- Girdi boru hatları oluşturmak ve verimli model eğitimi sağlamak için PyTorch'un Veri Kümesi ve DataLoader ile çalışma
- Optimize edilmiş makine öğrenimi kodu yazmak için PyTorch ile çalışma
- Yaygın derin öğrenme mimarilerini rahatça uygulamak için torch.nn modülünü kullanma
- Yapay NN'ler için aktivasyon fonksiyonlarının seçilmesi

PyTorch ve eğitim performansı

PyTorch, makine öğrenimi görevlerimizi önemli ölçüde hızlandırabilir. Bunu nasıl yapabileceğini anlamak için, donanımımızda pahalı hesaplamalar yaparken genellikle karşılaştığımız bazı performans zorluklarını tartışarak başlayalım. Ardından, PyTorch'un ne olduğuna ve bu bölümde öğrenme yaklaşımımızın ne olacağına üst düzey bir bakış atacağız.

Performans zorlukları

Bilgisayar işlemcilerinin performansı elbette son yıllarda sürekli olarak gelişmektedir. Bu da daha güçlü ve karmaşık öğrenme sistemlerini eğitmemize olanak tanıyor; bu da makine öğrenimi modellerimizin tahmin performansını artırabileceğimiz anlamına geliyor. Şu anda mevcut olan en ucuz masaüstü bilgisayar donanımı bile birden fazla çekirdeğe sahip işlem birimleriyle birlikte geliyor.

Önceki bölümlerde, scikit-learn'deki birçok fonksiyonun bu komplikasyonları birden fazla işlem birimine yaymamıza izin verdiğini gördük. Ancak varsayılan olarak Python, **global yorumlayıcı kilidi** (**GIL**) nedeniyle tek bir çekirdek üzerinde yürütme ile sınırlıdır. Dolayısıyla, hesaplamalarımızı birden fazla çekirdeğe dağıtmak için Python'un çoklu işlem kütüphanesinden yararlansak da, en gelişmiş masaüstü donanımının nadiren 8 veya 16'dan fazla çekirdeğe sahip olduğunu göz önünde bulundurmamız gerekir.

Sıfırdan Çok Katmanlı Yapay Sinir Ağı Uygulaması başlıklı *11. Bölüm'*den hatırlayacağınız üzere, 100 birimden oluşan tek bir gizli katmana sahip **çok** basit bir **çok katmanlı algılayıcı** (**MLP**) uygulamıştık. Çok basit bir görüntü sınıflandırma görevi için yaklaşık 80.000 ağırlık parametresini ([784*100 + 100] + [100 * 10] + 10 = 79.510) optimize etmek zorunda kaldık. MNIST'teki görüntüler oldukça küçüktür (28×28) ve ek gizli katmanlar eklemek veya daha yüksek piksel yoğunluklarına sahip görüntülerle çalışmak istersek parametre sayısındaki patlamayı ancak hayal edebiliriz. Böyle bir görev tek bir işlem birimi için hızla olanaksız hale gelecektir. O zaman soru şu oluyor: Bu tür sorunların üstesinden nasıl daha etkili bir şekilde gelebiliriz?

Bu sorunun bariz çözümü, gerçek birer iş atı olan **grafik işlem birimlerini** (**GPU'lar**) kullanmaktır. Bir grafik kartını makinenizin içindeki küçük bir bilgisayar kümesi olarak düşünebilirsiniz. Bir başka avantajı da, aşağıdaki genel bakışta görebileceğiniz gibi, modern GPU'ların son teknoloji **merkezi işlem birimlerine** (**CPU'lar)** kıyasla çok değerli olmasıdır:

Specifications	Intel® Core™ i9-11900KB Processor	NVIDIA GeForce® RTX™ 3080 Ti
Base Clock Frequency	3.3 GHz	1.37 GHz
Cores	16 (32 threads)	10240
Memory Bandwidth	45.8 GB/s	912.1 GB/s
Floating-Point Calculations	742 GFLOPS	34.10 TFLOPS
Cost	~ $540.00	~ $1200.00

Şekil 12.1: Son teknoloji bir CPU ve GPU'nun karşılaştırılması

*Şekil 12.1'*deki bilgilerin kaynakları aşağıdaki web siteleridir (erişim tarihi: Temmuz 2021):

- https://ark.intel.com/content/www/us/en/ark/products/215570/intel-core-i9-11900kb-processor-24m-cache-up-to-4-90-ghz.html
- https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3080-3080ti/

Modern bir CPU'nun 2,2 katı fiyatına, 640 kat daha fazla çekirdeğe sahip ve saniyede yaklaşık 46 kat daha fazla kayan nokta hesaplaması yapabilen bir GPU alabiliyoruz. Peki, makine öğrenimi görevlerimiz için GPU'ları kullanmaktan bizi alıkoyan nedir? Buradaki zorluk, GPU'ları hedeflemek için kod yazmanın Python kodunu yorumlayıcımızda çalıştırmak kadar basit olmamasıdır. GPU'yu hedeflememizi sağlayan CUDA ve OpenCL gibi özel paketler vardır. Ancak, CUDA veya OpenCL'de kod yazmak muhtemelen makine öğrenimi algoritmalarını uygulamak ve çalıştırmak için en uygun yol değildir. İyi haber şu ki PyTorch bunun için geliştirildi!

 alinpikbarbell@gmail.com

PyTorch nedir?

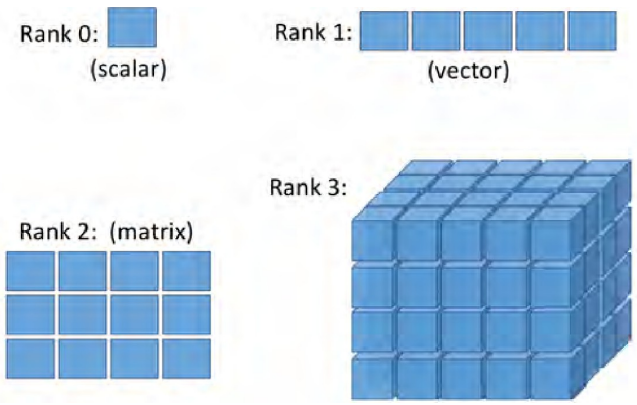
PyTorch, derin öğrenme için uygun sarmalayıcılar da dahil olmak üzere, ma- chine öğrenme algoritmalarını uygulamak ve çalıştırmak için ölçeklenebilir ve çok platformlu bir programlama arayüzüdür. PyTorch öncelikle **Facebook AI Research** (**FAIR**) laboratuvarındaki araştırmacılar ve mühendisler tarafından geliştirilmiştir. Geliştirilmesinde topluluktan da birçok katkı bulunmaktadır. PyTorch ilk olarak Eylül 2016'da piyasaya sürülmüştür ve değiştirilmiş BSD lisansı altında ücretsiz ve açık kaynaklıdır. Akademi ve endüstriden birçok makine öğrenimi araştırmacısı ve uygulayıcısı, Tesla Autopilot, Uber's Pyro ve Hugging Face's Transformers gibi derin öğrenme çözümleri geliştirmek için PyTorch'u uyarlamıştır (https://pytorch. org/ecosystem/).

PyTorch, makine öğrenimi modellerinin eğitim performansını artırmak için CPU'larda, GPU'larda ve TPU'lar gibi XLA cihazlarında yürütmeye izin verir. Bununla birlikte, en yüksek performans yetenekleri GPU'lar ve XLA cihazları kullanıldığında keşfedilebilir. PyTorch, CUDA özellikli ve ROCm GPU'ları resmi olarak desteklemektedir. PyTorch'un geliştirilmesi Torch kütüphanesine dayanmaktadır (www.torch.ch). Adından da anlaşılacağı gibi, Python arayüzü PyTorch'un birincil geliştirme odağıdır.

PyTorch, bir dizi düğümden oluşan bir hesaplama grafiği etrafında inşa edilmiştir. Her düğüm, sıfır veya daha fazla girdisi veya çıktısı olabilen bir işlemi temsil eder. PyTorch, işlemleri değerlendiren, hesaplamayı yürüten ve somut değerleri hemen döndüren zorunlu bir programlama ortamı sağlar. Bu nedenle, PyTorch'taki hesaplama grafiği önceden oluşturulmak ve daha sonra yürütülmek yerine örtük olarak tanımlanır.

Matematiksel olarak, tensörler skalerlerin, vektörlerin, matrislerin ve benzerlerinin bir genellemesi olarak anlaşılabilir. Daha somut olarak, bir skaler rank-0 tensörü olarak tanımlanabilir, bir vektör rank-1 tensörü olarak tanımlanabilir, bir matris rank-2 tensörü olarak tanımlanabilir ve üçüncü bir boyutta istiflenmiş matrisler rank-3 tensörleri olarak tanımlanabilir. PyTorch'taki tensörler NumPy'nin dizilerine benzer, ancak tensörler otomatik farklılaştırma için optimize edilmiştir ve GPU'larda çalışabilir.

Tensör kavramını daha açık hale getirmek için, ilk sırada 0 ve 1 dereceli tensörleri ve ikinci sırada 2 ve 3 dereceli tensörleri temsil eden *Şekil 12.2'yi* düşünün:



Şekil 12.2: PyTorch'ta farklı tensör türleri

Artık PyTorch'un ne olduğunu bildiğimize göre, nasıl kullanılacağını görelim.

PyTorch'u nasıl öğreneceğiz

İlk olarak, PyTorch'un programlama modelini, özellikle de tensör oluşturma ve manipüle etmeyi ele alacağız. Ardından, verileri nasıl yükleyeceğimizi ve bir veri kümesi üzerinde verimli bir şekilde yineleme yapmamızı sağlayacak torch.utils.data modülünü nasıl kullanacağımızı göreceğiz. Buna ek olarak, torch.utils.data.Dataset alt modülündeki mevcut, kullanıma hazır veri kümelerini tartışacağız ve bunları nasıl kullanacağımızı öğreneceğiz.

Bu temel bilgileri öğrendikten sonra PyTorch sinir ağı torch.nn modülü tanıtılacaktır. Daha sonra, makine öğrenimi modelleri oluşturmaya geçeceğiz, modelleri nasıl oluşturacağımızı ve eğiteceğimizi öğreneceğiz ve eğitilmiş modelleri gelecekte değerlendirmek üzere diske nasıl kaydedeceğimizi öğreneceğiz.

PyTorch ile ilk adımlar

Bu bölümde, düşük seviyeli PyTorch API'sini kullanmak için ilk adımlarımızı atacağız. PyTorch'u kurduktan sonra, PyTorch'ta tensörlerin nasıl oluşturulacağını ve şekillerini, veri türlerini vb. değiştirmek gibi farklı manipülasyon yollarını ele alacağız.

PyTorch'u Yükleme

PyTorch'u kurmak için resmi https://pytorch.org web sitesindeki en son talimatlara başvurmanızı öneririz. Aşağıda, çoğu sistemde çalışacak temel adımları özetleyeceğiz.

Sisteminizin nasıl kurulduğuna bağlı olarak, genellikle Python'un pip yükleyicisini kullanabilir ve terminalinizden aşağıdakileri çalıştırarak PyPI'dan PyTorch'u yükleyebilirsiniz:

```
pip install torch torchvision
```

Bu, yazım sırasında 1.9.0 olan en son *kararlı* sürümü yükleyecektir. Aşağıdaki kod örnekleriyle uyumlu olması garanti edilen 1.9.0 sürümünü yüklemek için, önceki komutu aşağıdaki gibi değiştirebilirsiniz:

```
pip install torch==1.9.0 torchvision==0.10.0
```

GPU'ları kullanmak istiyorsanız (önerilir), CUDA ve cuDNN'yi destekleyen uyumlu bir NVIDIA grafik kartına ihtiyacınız vardır. Makineniz bu gereksinimleri karşılıyorsa, PyTorch'u GPU desteği ile aşağıdaki gibi yükleyebilirsiniz:

```
pip install torch==1.9.0+cu111 torchvision==0.10.0+cu111 -f https://download.
pytorch.org/whl/torch_stable.html
```

CUDA 11.1 için veya:

```
pip install torch==1.9.0 torchvision==0.10.0\ -f https://download.pytorch.org/
whl/torch_stable.html
```

CUDA 10.2 için yazım zamanı itibariyle.

macOS ikili dosyaları CUDA'yı desteklemediği için kaynaktan yükleyebilirsiniz: https://pytorch.org/get- started/locally/#mac-from-source.

Kurulum ve ayarlama süreci hakkında daha fazla bilgi için lütfen https://pytorch.org/get-started/locally/ adresindeki resmi önerilere

olimpikbarbell@gmail.com

bakın.

PyTorch'un aktif olarak geliştirilmekte olduğunu unutmayın; bu nedenle, her birkaç ayda bir önemli değişikliklerle yeni sürümler yayınlanmaktadır. PyTorch sürümünüzü terminalinizden aşağıdaki gibi

```
python -c 'import torch; print(torch. version )'
```

PyTorch kurulumunuzda sorun giderme

Kurulum prosedürüyle ilgili sorun yaşarsanız, https://pytorch.org/ get-started/locally/ adresinde sağlanan sistem ve platforma özgü öneriler hakkında daha fazla bilgi edinin. Bu bölümdeki tüm kodların CPU'nuzda çalıştırılabileceğini unutmayın; bir GPU kullanmak tamamen isteğe bağlıdır, ancak PyTorch'un avantajlarından tam olarak yararlanmak istiyorsanız önerilir. Örneğin, bazı NN modellerinin CPU üzerinde eğitilmesi bir hafta sürebilirken, aynı modeller modern bir GPU üzerinde sadece birkaç saat içinde eğitilebilir. Bir grafik kartınız varsa, uygun şekilde ayarlamak için kurulum sayfasına bakın. Ayrıca, Ubuntu'da NVIDIA grafik kartı sürücülerinin, CUDA'nın ve cuDNN'nin nasıl kurulacağını açıklayan bu kurulum kılavuzunu yararlı bulabilirsiniz (gerekli değildir ancak PyTorch'u GPU'da çalıştırmak için önerilen gereksinimlerdir): https://sebastianraschka.com/pdf/books/dlb/ appendix_h_cloud-computing.pdf. Ayrıca, *Bölüm 17, Yeni Verilerin Sentezlenmesi için Üretken Çekişmeli Ağlar*'da göreceğiniz gibi, modellerinizi Google Colab aracılığıyla ücretsiz olarak bir GPU kullanarak da eğitebilirsiniz.

doğrulayabilirsiniz:

PyTorch'ta tensör oluşturma

Şimdi, tensör oluşturmanın birkaç farklı yolunu ele alalım ve ardından bazı özelliklerini ve bunları nasıl manipüle edeceğimizi görelim. İlk olarak, aşağıdaki gibi torch.tensor veya torch.from_numpy fonksiyonunu kullanarak bir listeden veya NumPy dizisinden bir tensör oluşturabiliriz:

```
>> import torch
>> import numpy as np
>>  np.set_printoptions(precision=3)
>>> a = [1, 2, 3]
>>> b = np.array([4, 5, 6], dtype=np.int32)
>>> t_a = torch.tensor(a)
>>> t_b = torch.from_numpy(b)
>> print(t_a)
>> print(t_b)
tensör([1, 2, 3])
tensor([4, 5, 6], dtype=torch.int32)
```

Bunun sonucunda t_a ve t_b tensörleri, shape=(3,) ve dtype=int32 özellikleri ile kaynaklarından uyarlandı. NumPy dizilerine benzer şekilde, bu özellikleri de görebiliriz:

```
>> t_ones = torch.ones(2, 3)
>> t_ones.shape
torch.Size([2, 3 ])
>> print(t_ones)
```

 alinpikbarbell@gmail.com

```
tensör([[1., 1., 1.],
        [1., 1., 1.]])
```

Son olarak, rastgele değerlerden oluşan bir tensör oluşturmak aşağıdaki şekilde yapılabilir:

```
>> rand_tensör = torch.rand(2,3)
>> print(rand_tensor)
tensor([[0.1409, 0.2848, 0.8914],
        [0.9223, 0.2924, 0.7889]])
```

Bir tensörün veri türünü ve şeklini değiştirme

Tensörleri manipüle etmenin yollarını öğrenmek, onları bir modele veya bir işleme giriş için uyumlu hale getirmek için gereklidir. Bu bölümde, tensör veri türlerini ve şekillerini döküm, yeniden şekillendirme, transpoze etme ve sıkıştırma (boyutları kaldırma) gibi çeşitli PyTorch fonksiyonları aracılığıyla nasıl manipüle edeceğinizi öğreneceksiniz.

Torch.to() fonksiyonu, bir tensörün veri tipini istenen bir tipe değiştirmek için kullanılabilir:

```
>> t_a_new = t_a.to(torch.int64)
>> print(t_a_new.dtype) torch.int64
```

Diğer tüm veri türleri için https://pytorch.org/docs/stable/tensor_attributes.html adresine bakın.

İlerleyen bölümlerde göreceğiniz gibi, bazı işlemler giriş tensörlerinin belirli sayıda elemanla (şekil) ilişkili belirli sayıda boyuta (yani rütbe) sahip olmasını gerektirir. Bu nedenle, bir tensörün şeklini değiştirmemiz, yeni bir boyut eklememiz veya gereksiz bir boyutu çıkarmamız gerekebilir. PyTorch bunu başarmak için torch.transpose(), torch.reshape() ve torch.squeeze() gibi faydalı fonksiyonlar (veya işlemler) sağlar. Şimdi bazı örneklere bir göz atalım:

- Bir tensörün aktarılması:

```
>> t = torch.rand(3, 5)
>>> t_tr = torch.transpose(t, 0, 1)
>> print(t.shape, ' --> ', t_tr.shape)
torch.Size([3, 5]) --> torch.Size([5, 3])
```

- Bir tensörü yeniden şekillendirme (örneğin, 1 boyutlu bir vektörden 2 boyutlu bir diziye):

```
>> t = torch.zeros(30)
>> t_reshape = t.reshape(5, 6)
>> print(t_reshape.shape) torch.Size([5,
6])
```

- Gereksiz boyutların kaldırılması (gerekli olmayan 1 boyutuna sahip boyutlar):

```
>> t = torch.zeros(1, 2, 1, 4, 1)
>> t_sqz = torch.squeeze(t, 2)
>> print(t.shape, ' --> ', t_sqz.shape)
torch.Size([1, 2, 1, 4, 1]) --> torch.Size([1 , 2, 4, 1 ])
```

Tensörlere matematiksel işlemler uygulama

Matematiksel işlemleri, özellikle de doğrusal cebir işlemlerini uygulamak, çoğu makine öğrenimi modelini oluşturmak için gereklidir. Bu alt bölümde, eleman-bilge çarpımı, matris çarpımı ve bir tensörün normunu hesaplama gibi yaygın olarak kullanılan bazı doğrusal cebir işlemlerini ele alacağız.

İlk olarak, biri [-1, 1) aralığında tekdüze dağılıma ve diğeri standart normal dağılıma sahip iki rastgele tensör örnekleyelim:

```
>>> torch.manual_seed(1)
>> t1 = 2 * torch.rand(5, 2) - 1
>> t2 = torch.normal(mean=0, std=1, size=(5, 2))
```

torch.rand'ın [0, 1) aralığında tekdüze bir dağılımdan rastgele sayılarla dolu bir tensör döndürdüğünü unutmayın.

t1 ve t2'nin aynı şekle sahip olduğuna dikkat edin. Şimdi, t1 ve t2'nin eleman-bilge çarpımını hesaplamak için aşağıdakileri kullanabiliriz:

```
>>> t3 = torch.multiply(t1, t2)
>> print(t3)
tensör([[ 0.4426, -0.3114],
        [ 0.0660, -0.5970],
        [ 1.1249, 0.0150],
        [ 0.1569, 0.7107],
        [-0.0451, -0.0352]])
```

Belirli bir eksen (veya eksenler) boyunca ortalama, toplam ve standart sapmayı hesaplamak için torch. mean(), torch.sum() ve torch.std()'yi kullanabiliriz. Örneğin, t1'deki her bir sütunun ortalaması aşağıdaki gibi hesaplanabilir:

```
>>> t4 = torch.mean(t1, axis=0)
>> print(t4)
tensör([-0.1373, 0 . 2028])
```

t1 ve t2 arasındaki matris-matris çarpımı (yani, $t_1 \times t_2^T$, burada üst simge T transpoze içindir) torch.matmul() işlevi kullanılarak aşağıdaki gibi hesaplanabilir:

```
>>> t5 = torch.matmul(t1, torch.transpose(t2, 0, 1))
>> print(t5)
tensör([ 0.1312, 0.3860, -0.6267, -1.0096, -0.2943],
       [ 0.1647, -0.5310,  0.2434,  0.8035,  0.1980],
       [-0.3855, -0.4422,  1.1399,  1.5558,  0.4781],
       [ 0.1822, -0.5771,  0.2585,  0.8676,  0.2132],
       [ 0.0330, 0.1084, -0.1692, -0.2771, -0.0804]])
```

 olimpikbarbell@gmail.com

Öte yandan, $t_1{}^T \times t_2$ hesaplaması t1'in transpoze edilmesiyle gerçekleştirilir ve 2×2 boyutunda bir dizi elde edilir:

```
>>> t6 = torch.matmul(torch.transpose(t1, 0, 1), t2)
>> print(t6)
tensör([[ 1.7453, 0.3392],
        [-1.6038, -0.2180]])
```

Son olarak, torch.linalg.norm() fonksiyonu bir tensörün L^p normunu hesaplamak için kullanışlıdır. Örneğin, t1'in L^2 normunu aşağıdaki gibi hesaplayabiliriz:

```
>> norm_t1 = torch.linalg.norm(t1, ord=2, dim=1)
>> print(norm_t1)
tensör([0.6785, 0.5078, 1.1162, 0.5488, 0.1853])
```

Bu kod parçacığının t1'in L^2 normunu doğru hesapladığını doğrulamak için sonuçları aşağıdaki NumPy işleviyle karşılaştırabilirsiniz: np.sqrt(np.sum(np.square(t1.numpy()), axis=1)).

Tensörleri bölme, yığma ve birleştirme

Bu alt bölümde, bir tensörü birden fazla tensöre bölmek için PyTorch işlemlerini veya tersini ele alacağız: birden fazla tensörü tek bir tensörde istifleme ve birleştirme.

Tek bir tensörümüz olduğunu ve bunu iki veya daha fazla tensöre bölmek istediğimizi varsayalım. Bunun için PyTorch, bir giriş tensörünü eşit boyutlu tensörlerin bir listesine bölen kullanışlı bir torch.chunk() işlevi sağlar. Bir tensörü dim bağımsız değişkeni tarafından belirtilen istenen boyut boyunca bölmek için chunks bağımsız değişkenini kullanarak istenen bölme sayısını bir tamsayı olarak belirleyebiliriz. Bu durumda, belirtilen boyut boyunca giriş tensörünün toplam boyutu istenen bölme sayısına bölünebilir olmalıdır. Alternatif olarak, torch.split() fonksiyonunu kullanarak istenen boyutları bir liste halinde sağlayabiliriz. Şimdi bu iki seçeneğin bir örneğine bakalım:

- Bölünme sayısını sağlar:

```
>>> torch.manual_seed(1)
>>> t = torch.rand(6)
>> print(t)
tensör([0.7576, 0.2793, 0.4031, 0.7347, 0.0293, 0.7999])
>> t_splits = torch.chunk(t, 3)
>> [item.numpy() for item in t_splits] [array([0.758,
0.279], dtype=float32),
 array([0.403, 0.735], dtype=float32),
 array([0.029, 0.8 ], dtype=float32)]
```

  Bu örnekte, 6 boyutlu bir tensör, her biri 2 boyutlu üç tensörden oluşan bir listeye bölünmüştür. Tensör boyutu yığın değerine bölünemiyorsa, son yığın daha küçük olacaktır.

- Farklı bölmelerin boyutlarının sağlanması:

  Alternatif olarak, bölme sayısını tanımlamak yerine, çıktı tensörlerinin boyutlarını doğrudan da belirtebiliriz. Burada, 5 boyutlu bir tensörü 3 ve 2 boyutlu tensörlere bölüyoruz:

```
>>> torch.manual_seed(1)
>>> t = torch.rand(5)
>> print(t)
tensör([0.7576, 0.2793, 0.4031, 0.7347, 0.0293])
>> t_splits = torch.split(t, split_size_or_sections=[3, 2])
>> [item.numpy() for item in t_splits] [array([0.758,
0.279, 0.403], dtype=float32),
 array([0.735, 0.029], dtype=float32)]
```

Bazen, birden fazla tensörle çalışırız ve tek bir tensör oluşturmak için bunları birleştirmemiz veya yığmamız gerekir. Bu durumda, torch.stack() ve torch.cat() gibi PyTorch fonksiyonları kullanışlı olur. Örneğin, 3 boyutlu 1'ler içeren bir 1D tensör, A ve 2 boyutlu 0'lar içeren bir 1D tensör, B oluşturalım ve bunları 5 boyutlu bir 1D tensör, C'de birleştirelim:

```
>>> A = torch.ones(3)
>>> B = torch.zeros(2)
>>> C = torch.cat([A, B], axis=0)
>> print(C)
tensör([1., 1., 1., 0., 0.])
```

Her ikisi de 3 boyutlu olan 1 boyutlu A ve B tensörleri oluşturursak, bunları bir araya getirerek 2 boyutlu S tensörünü oluşturabiliriz:

```
>>> A = torch.ones(3)
>>> B = torch.zeros(3)
>>> S = torch.stack([A, B], axis=1)
>> print(S)
tensör([[1., 0.],
        [1., 0.],
        [1., 0.]])
```

PyTorch API, bir model oluşturmak, verilerinizi işlemek ve daha fazlası için kullanabileceğiniz birçok işleme sahiptir. Ancak, her işlevi ele almak bu kitabın kapsamı dışındadır, burada en önemli olanlara odaklanacağız. İşlemlerin ve fonksiyonların tam listesi için PyTorch'un https://pytorch.org/docs/stable/index.html adresindeki dokümantasyon sayfasına başvurabilirsiniz.

 ilinpikbarbell@gmail.com

PyTorch'ta girdi boru hatları oluşturma

Bir derin NN modelini eğitirken, önceki bölümlerde gördüğümüz gibi, genellikle stokastik gradyan inişi gibi yinelemeli bir optimizasyon algoritması kullanarak modeli aşamalı olarak eğitiriz.

Bu bölümün başında belirtildiği gibi, torch.nn NN modelleri oluşturmak için kullanılan bir modüldür. Eğitim veri kümesinin oldukça küçük olduğu ve belleğe bir tensör olarak yüklenebildiği durumlarda, bu tensörü eğitim için doğrudan kullanabiliriz. Ancak tipik kullanım durumlarında, veri kümesi bilgisayar belleğine sığmayacak kadar büyük olduğunda, verileri ana depolama aygıtından (örneğin sabit sürücü veya katı hal sürücüsü) parçalar halinde, yani yığın yığın yüklememiz gerekecektir. (PyTorch terminolojisine yakın kalmak için bu bölümde "mini-batch" yerine "batch" teriminin kullanıldığına dikkat edin). Ayrıca, eğitim prosedürünü güçlendirmek ve aşırı uyumu önlemek için ortalama merkezleme, ölçekleme veya gürültü ekleme gibi belirli dönüşümleri ve ön işleme adımlarını verilerimize uygulamak için bir veri işleme boru hattı oluşturmamız gerekebilir.

Ön işleme işlevlerini her seferinde manuel olarak uygulamak oldukça zahmetli olabilir. Neyse ki PyTorch, verimli ve kullanışlı ön işleme boru hatları oluşturmak için özel bir sınıf sağlar. Bu bölümde, bir PyTorch Veri Kümesi ve DataLoader oluşturmak ve veri yükleme, karıştırma ve gruplama uygulamak için farklı yöntemlere genel bir bakış göreceğiz.

Mevcut tensörlerden bir PyTorch DataLoader oluşturma

Veriler zaten bir tensör nesnesi, bir Python listesi veya bir NumPy dizisi biçiminde mevcutsa, torch.utils.data.DataLoader() sınıfını kullanarak kolayca bir veri kümesi yükleyicisi oluşturabiliriz. DataLoader sınıfının bir nesnesini döndürür ve bu nesneyi giriş veri kümesindeki tek tek öğeler arasında yineleme yapmak için kullanabiliriz. Basit bir örnek olarak, 0'dan 5'e kadar olan değerlerin

```
>> from torch.utils.data import DataLoader
>>> t = torch.arange(6, dtype=torch.float32)
>> data_loader = DataLoader(t)
```

listesinden bir veri kümesi oluşturan aşağıdaki kodu düşünün:

Bir veri kümesini aşağıdaki gibi giriş giriş kolayca yineleyebiliriz:

```
>> for item in data_loader:
...     print(öğe)
tensör([0.])
tensör([1.])
tensör([2.])
tensör([3.])
tensör([4.])
tensör([5.])
```

Bu veri kümesinden, istenen parti boyutu 3 olacak şekilde partiler oluşturmak istiyorsak, bunu batch_size argümanını aşağıdaki gibi kullanın:

```
>> data_loader = DataLoader(t, batch_size=3, drop_last=False)
>> for i, batch in enumerate(data_loader, 1):
...     print(f'batch {i}:', batch)
yığın 1: tensör([0., 1., 2.])
yığın 2: tensör([3., 4., 5.])
```

Bu, bu veri kümesinden iki grup oluşturacaktır; ilk üç öğe grup #1'e ve kalan öğeler grup #2'ye gidecektir. İsteğe bağlı drop_last bağımsız değişkeni, tensördeki öğe sayısının istenen yığın boyutuna bölünemediği durumlar için kullanışlıdır. drop_last öğesini True olarak ayarlayarak tam olmayan son yığını düşürebiliriz. drop_last için varsayılan değer False'dir.

Bir veri kümesini her zaman doğrudan yineleyebiliriz, ancak az önce gördüğünüz gibi DataLoader bir veri kümesine otomatik ve özelleştirilebilir bir yığınlama sağlar.

İki tensörün ortak bir veri kümesinde birleştirilmesi

Genellikle, veriler iki (veya muhtemelen daha fazla) tensörde olabilir. Örneğin, özellikler için bir tensöre ve etiketler için bir tensöre sahip olabiliriz. Bu gibi durumlarda, bu tensörleri birleştiren ve bu tensörlerin öğelerini tuple'lar halinde almamızı sağlayacak bir veri kümesi oluşturmamız gerekir.

t_x ve t_y olmak üzere iki tensörümüz olduğunu varsayalım. Tensör t_x, her biri 3 boyutunda olan özellik değerlerimizi tutar ve
t_y sınıf etiketlerini saklar. Bu örnek için ilk olarak bu iki tensörü aşağıdaki gibi oluşturuyoruz:

```
>>> torch.manual_seed(1)
>>> t_x = torch.rand([4, 3], dtype=torch.float32)
>> t_y = torch.arange(4)
```

Şimdi, bu iki tensörden ortak bir veri kümesi oluşturmak istiyoruz. Öncelikle aşağıdaki gibi bir Dataset sınıfı oluşturmamız gerekiyor:

```
>> from torch.utils.data import Dataset
>> class JointDataset(Dataset):
...     def init (self, x, y):
...         self.x = x
...         self.y = y
...
...     def len (self):
...         return len(self.x)
...
```

alinpikbarbell@gmail.com

```
...     def getitem (self, idx):
...         return self.x[idx], self.y[idx]
```

Özel bir Dataset sınıfı, daha sonra veri yükleyici tarafından kullanılmak üzere aşağıdaki yöntemleri içermelidir:

- init (): Mevcut dizilerin okunması, bir dosyanın yüklenmesi, verilerin filtrelenmesi gibi ilk mantığın gerçekleştiği yer burasıdır.
- getitem (): Bu, verilen indekse karşılık gelen örneği döndürür. Daha sonra

aşağıdaki gibi özel Dataset sınıfı ile t_x ve t_y'nin ortak bir veri kümesini oluştururuz:

```
>> from torch.utils.data import TensorDataset
>> joint_dataset = TensorDataset(t_x, t_y)
```

Son olarak, ortak veri kümesinin her bir örneğini aşağıdaki gibi yazdırabiliriz:

```
>> örneğin joint_dataset içinde:
...     print(' x: ', örnek[0], ' y: ', örnek[1])
  x:  tensör([0.7576, 0.2793, 0.4031])   y:  tensör(0)
  x:  tensör([0.7347, 0.0293, 0.7999])   y:  tensör(1)
  x:  tensör([0.3971, 0.7544, 0.5695])   y:  tensör(2)
  x:  tensör([0.4388, 0.6387, 0.5247])   y:  tensör(3)
```

İkinci veri kümesi tensör biçiminde etiketli bir veri kümesi ise torch.utils.data.TensorDataset sınıfını da kullanabiliriz. Böylece, kendi tanımladığımız Dataset sınıfımız olan JointDataset'i kullanmak yerine, aşağıdaki gibi bir ortak veri kümesi oluşturabiliriz:

```
>> joint_dataset = TensorDataset(t_x, t_y)
```

Yaygın bir hata kaynağının, orijinal özellikler (x) ve etiketler (y) arasındaki eleman-bilge uyumunun kaybolması olabileceğini unutmayın (örneğin, iki veri kümesi ayrı ayrı karıştırılırsa). Ancak, tek bir veri kümesinde birleştirildiklerinde, bu işlemleri uygulamak güvenlidir.

Diskteki görüntü dosya adları listesinden oluşturulmuş bir veri kümemiz varsa, görüntüleri bu dosya adlarından yüklemek için bir işlev tanımlayabiliriz. Bu bölümün ilerleyen kısımlarında bir veri kümesine birden fazla dönüşüm uygulamanın bir örneğini göreceksiniz.

Karıştır, grupla ve tekrarla

Bölüm 2, Sınıflandırma için Basit Makine Öğrenimi Algoritmalarının Eğitimi'nde belirtildiği gibi, stokastik gradyan inişi optimizasyonunu kullanarak bir NN modelini eğitirken, eğitim verilerini rastgele karıştırılmış gruplar halinde beslemek önemlidir. Bir veri yükleyici nesnesinin batch_size bağımsız değişkenini kullanarak yığın boyutunu nasıl belirleyeceğinizi daha önce görmüştünüz. Şimdi, yığın oluşturmaya ek olarak, veri kümeleri üzerinde nasıl karıştırma ve tekrarlama yapılacağını göreceksiniz. Önceki ortak veri kümesi ile çalışmaya devam edeceğiz.

İlk olarak, joint_dataset veri kümesinden bir karıştırılmış sürüm veri yükleyicisi oluşturalım:

```
>>> torch.manual_seed(1)
>> data_loader = DataLoader(dataset=joint_dataset, batch_size=2, shuffle=True)
```

Burada, her parti iki veri kaydı (x) ve bunlara karşılık gelen etiketler (y) içerir. Şimdi veri yükleyiciyi aşağıdaki gibi giriş giriş yineleyeceğiz:

```
>> for i, batch in enumerate(data_loader, 1):
...     print(f'batch {i}:', 'x:', batch[0],
              '\n            y:', batch[1])
b a t c h   1: x: tensor([[0.4388, 0.6387,
0.5247],
         [0.3971, 0.7544, 0.5695]])
          y: tensör([3, 2])
parti 2: x: tensör([[0.7576, 0.2793, 0.4031],
         [0.7347, 0.0293, 0.7999]])
          y: tensör([0, 1])
```

Satırlar, x ve y'deki girdiler arasındaki bire bir yazışmayı kaybetmeden karıştırılır.

Buna ek olarak, bir modeli birden fazla epok için eğitirken, veri kümesini karıştırmamız ve istenen epok sayısına göre yinelememiz gerekir. Bu yüzden, harmanlanmış veri kümesi üzerinde iki kez

```
>> for epoch in range(2):
>>>     print(f'epoch {epoch+1}')
>>>     for i, batch in enumerate(data_loader, 1):
...         print(f'batch {i}:', 'x:', batch[0],
                  '\n            y:', batch[1])
dönem 1
parti 1: x: tensör([[0.7347, 0.0293, 0.7999],
         [0.3971, 0.7544, 0.5695]])
          y: tensör([1, 2])
parti 2: x: tensör([[0,4388, 0,6387, 0,5247],
         [0.7576, 0.2793, 0.4031]])
          y: tensör([3, 0])
dönem 2
yığın 1: x: tensör([[0,3971, 0,7544, 0,5695],
         [0.7576, 0.2793, 0.4031]])
          y: tensör([2, 0])
parti 2: x: tensör([[0.7347, 0.0293, 0.7999],
         [0.4388, 0.6387, 0.5247]])
          y: tensör([1, 3])
```

yineleme yapalım:

 olimpikbarbell@gmail.com

Bu, iki farklı grup kümesiyle sonuçlanır. İlk epokta, ilk parti bir çift değer [y=1, y=2] içerir ve ikinci parti bir çift değer [y=3, y=0] içerir. İkinci epokta, iki yığın sırasıyla [y=2, y=0] ve [y=1, y=3] değer çiftlerini içerir. Her iterasyon için, bir yığın içindeki elemanlar da karıştırılır.

Yerel depolama diskinizdeki dosyalardan bir veri kümesi oluşturma

Bu bölümde, diskte depolanan görüntü dosyalarından bir veri kümesi oluşturacağız. Bu bölümün çevrimiçi içeriğiyle ilişkili bir resim klasörü bulunmaktadır. Klasörü indirdikten sonra, JPEG formatında altı kedi ve köpek resmi görebilmeniz gerekir.

Bu küçük veri kümesi, saklanan dosyalardan bir veri kümesi oluşturmanın genel olarak nasıl çalıştığını gösterecektir. Bunu başarmak için iki ek modül kullanacağız: Görüntü dosyası içeriğini okumak için PIL'de Image ve ham içeriğin kodunu çözmek ve görüntüleri yeniden boyutlandırmak için torchvision'da transforms.

> PIL.Image ve torchvision.transforms modülleri, kitabın kapsamı dışında kalan birçok ek ve yararlı işlev sağlar. Bu işlevler hakkında daha fazla bilgi edinmek için resmi belgelere göz atmanız önerilir:
>
> PIL.Image için https://pillow.readthedocs.io/en/stable/reference/Image.html
>
> https://pytorch.org/vision/stable/transforms.html for torchvision.transforms

Başlamadan önce, bu dosyaların içeriğine bir göz atalım. Görüntü dosyalarının bir listesini oluşturmak için pathlib kütüphanesini kullanacağız:

```
>> import pathlib
>> imgdir_path = pathlib.Path('cat_dog_images')
>>> dosya_listesi = sorted([str(yol) for yol in
... imgdir_path.glob('*.jpg')])
>> print(dosya_listesi)
['cat_dog_images/dog-03.jpg', 'cat_dog_images/cat-01.jpg', ' cat_dog_images/cat-
02.jpg', 'cat_dog_images/cat-03.jpg', ' cat_dog_images/dog-01.jpg', ' cat_dog_
images/dog-02.jpg']
```

Daha sonra, bu görüntü örneklerini Matplotlib kullanarak görselleştireceğiz:

```
>> import matplotlib.pyplot as plt
>> import os
>>> from PIL import Görüntü
>>> fig = plt.figure(figsize=(10, 5))
>> for i, file in enumerate(file_list):
...         img = Image.open(dosya)
...         print('Resim şekli:', np.array(img).shape)
...         ax = fig.add_subplot(2, 3, i+1)
...         ax.set_xticks([]); ax.set_yticks([])
```

```
...         ax.imshow(img)
...         ax.set_title(os.path.basename(dosya),
            size=15)
>> plt.tight_layout()
>> plt.show()
Görüntü şekli: (900, 1200, 3)
Görüntü şekli: (900, 1200, 3)
Görüntü şekli: (900, 1200, 3)
Görüntü şekli: (900, 742, 3)
Görüntü şekli: (800, 1200, 3)
Görüntü şekli: (800, 1200, 3)
```

Şekil 12.3 örnek görüntüleri göstermektedir:

Şekil 12.3: Kedi ve köpek resimleri

Sadece bu görselleştirme ve basılı görüntü şekillerinden bile görüntülerin farklı en-boy oranlarına sahip olduğunu görebiliriz. Bu görüntülerin en boy oranlarını (veya veri dizisi şekillerini) yazdırırsanız, bazı görüntülerin 900 piksel yüksekliğinde ve 1200 piksel genişliğinde (900×1200), bazılarının 800×1200 ve birinin 900×742 olduğunu göreceksiniz. Daha sonra bu görüntüleri tutarlı bir boyuta getirmek için ön işleme tabi tutacağız. Dikkate alınması gereken bir diğer nokta da bu görüntülerin etiketlerinin dosya adları içinde verilmiş olmasıdır. Dolayısıyla, bu etiketleri dosya

```
>>> etiketler = [1 if 'dog' in
...                os.path.basename(dosya) else 0
...                        dosya_listesindeki dosya için]
>> print(etiketler)
[0, 0, 0, 1, 1, 1 ]
```

adları listesinden çıkarıyoruz ve köpeklere 1, kedilere 0 etiketi atıyoruz:

Şimdi elimizde iki liste var: dosya adlarının (veya her bir görüntünün yollarının) bir listesi ve etiketlerinin bir listesi. Önceki bölümde, iki diziden ortak bir veri kümesinin nasıl oluşturulacağını öğrendiniz. Burada, aşağıdakileri yapacağız:

```
>> class ImageDataset(Dataset):
...         def init (self, file_list, labels):
...             self.file_list = file_list
```

 alinpikbarbell@gmail.com

```
...             self.labels = etiketler
...
...         def getitem (self, index): file
...             = self.file_list[index] label
...             = self.labels[index] return
...             file, label
...
...         def len (self):
...             return len(self.labels)

>> image_dataset = ImageDataset(file_list, labels)
>> for file, label in image_dataset:
...         print(dosya, etiket)


cat_dog_images/cat-01.jpg 0
cat_dog_images/cat-02.jpg 0
cat_dog_images/cat-03.jpg 0
cat_dog_images/dog-01.jpg 1
cat_dog_images/dog-02.jpg 1
cat_dog_images/dog-03.jpg 1
```

Ortak veri kümesi dosya adlarına ve etiketlere sahiptir.

Daha sonra, bu veri kümesine dönüşümler uygulamamız gerekir: görüntü içeriğini dosya yolundan yükleyin, ham içeriği koddan arındırın ve örneğin 80×120 gibi istenen bir boyuta yeniden boyutlandırın. Daha önce de belirtildiği gibi, görüntüleri yeniden boyutlandırmak ve yüklenen pikselleri aşağıdaki gibi tensörlere dönüştürmek için torchvision.transforms modülünü

```
>> import torchvision.transforms as transforms
>> img_height, img_width = 80, 120
>> transform = transforms.Compose([
...         transforms.ToTensor(),
...         transforms.Resize((img_height, img_width)),
... ])
```

kullanıyoruz:

Şimdi ImageDataset sınıfını az önce tanımladığımız dönüşümle güncelliyoruz:

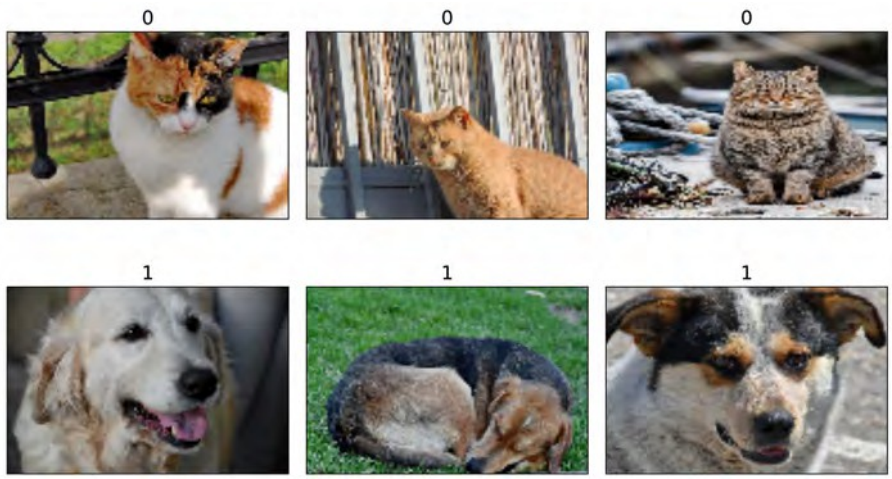
```
>> class ImageDataset(Dataset):
...         def init (self, file_list, labels, transform=None):
...             self.file_list = file_list
...             self.labels = labels
...             self.transform = transform
...
```

```
...     def getitem (self, index):
...         img = Image.open(self.file_list[index])
...         if self.transform is not None:
...             img = self.transform(img)
...         label = self.labels[index]
...         img, etiket döndür
...
...     def len (self):
...         return len(self.labels)
>>>
>> image_dataset = ImageDataset(file_list, labels, transform)
```

Son olarak, bu dönüştürülmüş görüntü örneklerini Matplotlib kullanarak görselleştiriyoruz:

```
>>> fig = plt.figure(figsize=(10, 6))
>> for i, example in enumerate(image_dataset):
...     ax = fig.add_subplot(2, 3, i+1)
...     ax.set_xticks([]); ax.set_yticks([])
...     ax.imshow(örnek[0].numpy().transpose((1, 2, 0)))
...     ax.set_title(f'{example[1]}', size=15)
...
>> plt.tight_layout()
>> plt.show()
```

Bu, alınan örnek görüntülerin etiketleriyle birlikte aşağıdaki şekilde görselleştirilmesiyle sonuçlanır:



Şekil 12.4: Görüntüler etiketlenir

 alinpikbarbell@gmail.com

ImageDataset sınıfındaki getitem yöntemi, ham içeriğin (görüntüler ve etiketler) yüklenmesi, görüntülerin tensörlere kodunun çözülmesi ve görüntülerin yeniden boyutlandırılması dahil olmak üzere dört adımı tek bir işlevde toplar. Fonksiyon daha sonra, üzerinde yineleme yapabileceğimiz ve önceki bölümlerde öğrendiğimiz shuffiing ve batching gibi diğer işlemleri bir veri yükleyici aracılığıyla uygulayabileceğimiz bir veri kümesi döndürür.

torchvision.datasets kütüphanesinden mevcut veri kümelerini getirme

torchvision.datasets kütüphanesi, derin öğrenme modellerini eğitmek veya değerlendirmek için ücretsiz olarak kullanılabilen görüntü veri kümelerinden oluşan güzel bir koleksiyon sağlar. Benzer şekilde, torchtext.datasets kütüphanesi doğal dil için veri kümeleri sağlar. Burada torchvision.datasets dosyasını örnek olarak kullanacağız.

Torchvision veri kümeleri (https://pytorch.org/vision/stable/datasets.html) güzel bir şekilde biçimlendirilmiştir ve özelliklerin ve etiketlerin biçimi, türü ve boyutluluğunun yanı sıra veri kümesinin orijinal kaynağına bağlantı da dahil olmak üzere bilgilendirici açıklamalarla birlikte gelir. Diğer bir avantaj ise bu veri kümelerinin hepsinin torch.utils.data.Dataset'in alt sınıfları olmasıdır, böylece önceki bölümlerde ele aldığımız tüm fonksiyonlar doğrudan kullanılabilir. Şimdi bu veri kümelerini nasıl kullanacağımızı görelim.

Öncelikle, torchvision'ı daha önce PyTorch ile birlikte yüklemediyseniz, torchvision kütüphanesini komut satırından pip aracılığıyla yüklemeniz gerekir:

```
pip install torchvision
```

Mevcut veri setlerinin listesine https://pytorch.org/vision/stable/datasets. html adresinden göz atabilirsiniz.

Aşağıdaki paragraflarda, iki farklı veri kümesinin getirilmesini ele alacağız: CelebA (celeb_a) ve MNIST rakam veri kümesi.

İlk olarak torchvision.datasets.CelebA (https://pytorch.org/vision/stable/datasets.html#celeba) ile CelebA veri kümesi (http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html) ile çalışalım. torchvision.datasets.CelebA'nın açıklaması, bu veri kümesinin yapısını anlamamıza yardımcı olacak bazı yararlı bilgiler sağlar:

- Veritabanının 'train', 'valid' ve 'test' olmak üzere üç alt kümesi vardır. Split parametresi ile belirli bir alt küme seçebilir veya hepsini yükleyebiliriz.
- Görüntüler PIL.Image formatında saklanır. Ve transforms.ToTensor ve transforms.Resize gibi özel bir dönüştürme işlevi kullanarak dönüştürülmüş bir sürüm elde edebiliriz.
- Kullanabileceğimiz 'öznitelikler', 'kimlik' ve 'yer işaretleri' gibi farklı hedef türleri vardır. 'nitelikler' görüntüdeki kişi için yüz ifadesi, makyaj, saç özellikleri vb. gibi 40 yüz niteliğidir; 'kimlik' bir görüntü için kişi kimliğidir; ve 'işaretler' gözlerin, burnun vb. konumu gibi çıkarılan yüz noktalarının sözlüğünü ifade eder.

Ardından, torchvision.datasets.CelebA sınıfını çağırarak verileri indirecek, diskte belirlenen bir klasörde saklayacak ve bir torch.utils.data.Dataset nesnesine yükleyeceğiz:

```
>> import torchvision
>> image_path = './'
>> celeba_dataset = torchvision.datasets.CelebA(
...         image_path, split='train', target_type='attr', download=True
... )
1443490838/? [01:28<00:00, 6730259.81it/s]
26721026/? [00:03<00:00, 8225581.57it/s]
3424458/? [00:00<00:00, 14141274.46it/s]
6082035/? [00:00<00:00, 21695906.49it/s]
12156055/? [00:00<00:00, 12002767.35it/s]
2836386/? [00:00<00:00, 3858079.93it/s]
```

BadZipFile: Dosya bir zip dosyası değil hatası veya RuntimeError ile karşılaşabilirsiniz: img_align_celeba.zip dosyasının günlük kotası aşıldı ve dosya indirilemiyor. Bu, Google Drive'ın bir sınırlamasıdır ve yalnızca daha sonra tekrar deneyerek aşılabilir; yalnızca Google Drive'ın CelebA dosyaları tarafından aşılan bir günlük maksimum kotası olduğu anlamına gelir. Bunu aşmak için dosyaları kaynaktan manuel olarak indirebilirsiniz: http://mmlab.ie.cuhk.edu.hk/projects/ CelebA.html. İndirilen celeba/ klasöründe img_align_celeba.zip dosyasını açabilirsiniz. image_path, indirilen celeba/ klasörünün köküdür. Dosyaları zaten bir kez indirdiyseniz, download=False olarak ayarlayabilirsiniz. Daha fazla bilgi ve rehberlik için https://github.com/rasbt/machine-learning-book/ blob/main/ch12/ch12_part1.ipynb adresindeki eşlik eden kod not defterine bakmanızı şiddetle tavsiye ederiz.

Veri kümelerini örneklediğimize göre, nesnenin torch.utils.data.Dataset türünde olup olmadığını kontrol edelim
Sınıf:

```
>> assert isinstance(celeba_dataset, torch.utils.data.Dataset)
```

Belirtildiği gibi, veri kümesi zaten eğitim, test ve doğrulama veri kümelerine ayrılmıştır ve biz yalnızca eğitim kümesini yüklüyoruz. Ve sadece 'attributes' hedefini kullanıyoruz. Veri örneklerinin neye benzediğini görmek için aşağıdaki kodu çalıştırabiliriz:

```
>> örnek = next(iter(celeba_dataset))
>> print(örnek)
(<PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=178x218 at
0x120C6C668>, tensor([0, 1 , 1, 0 , 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 , 1,
1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1 , 0, 1, 0 ,  1 ,  0 ,  0, 1 ]))
```

Bu veri kümesindeki örneğin (PIL.Image, attributes) şeklinde bir tuple olarak geldiğini unutmayın. Bu veri kümesini eğitim sırasında denetimli bir derin öğrenme modeline aktarmak istiyorsak, bunu (özellikler tensörü, etiket) şeklinde yeniden biçimlendirmemiz gerekir. Etiket için, 31. öğe olan özniteliklerden 'Gülümseyen' kategorisini örnek olarak kullanacağız.

 alinpikbarbell@gmail.com

Son olarak, 'Gülümseyen' etiketleriyle görselleştirmek için ilk 18 örneği ele alalım:

```
>> from itertools import islice
>>> fig = plt.figure(figsize=(12, 8))
>> for i, (image, attributes) in islice(enumerate(celeba_dataset), 18):
...         ax = fig.add_subplot(3, 6, i+1)
...         ax.set_xticks([]); ax.set_yticks([])
...         ax.imshow(görüntü)
...         ax.set_title(f'{attributes[31]}', size=15)
>> plt.show()
```

celeba_dataset'ten alınan örnekler ve etiketleri *Şekil 12.5*'te gösterilmektedir:

Şekil 12.5: Model gülümseyen ünlüleri tahmin ediyor

CelebA görüntü veri kümesini almak ve kullanmak için yapmamız gereken tek şey buydu.

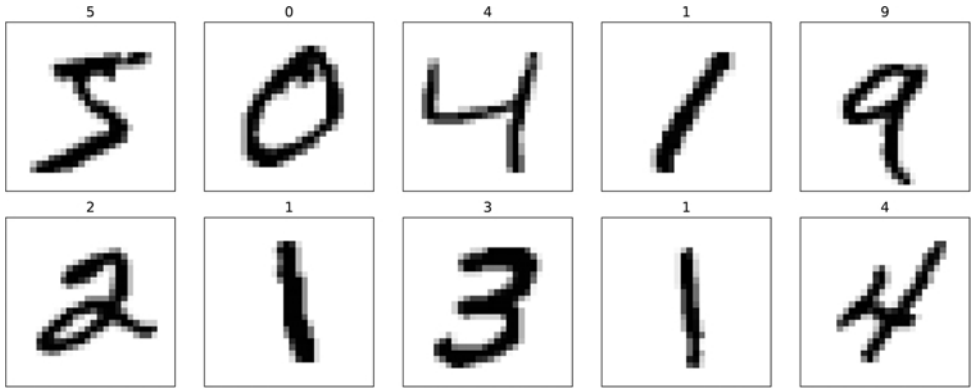
Daha sonra, torchvision.datasets.MNIST'ten ikinci veri setiyle devam edeceğiz (https://pytorch.org/vision/stable/datasets.html#mnist). MNIST rakam veri setini almak için nasıl kullanılabileceğini görelim:

- Veritabanının 'train' ve 'test' olmak üzere iki bölümü vardır. Yüklemek için belirli bir alt küme seçmemiz gerekiyor.
- Görüntüler PIL.Image formatında saklanır. Ve transforms.ToTensor ve transforms.Resize gibi özel bir dönüştürme işlevi kullanarak dönüştürülmüş bir sürüm elde edebiliriz.
- Hedef için 0'dan 9'a kadar 10 sınıf bulunmaktadır.

Şimdi, 'train' bölümünü indirebilir, öğeleri tuple'lara dönüştürebilir ve 10 örneği görselleştirebiliriz:

```
>> mnist_dataset = torchvision.datasets.MNIST(image_path, 'train',
download=True)
>> assert isinstance(mnist_dataset, torch.utils.data.Dataset)
> >  örnek = next(iter(mnist_dataset))
>> print(örnek)
(<PIL.Image.Image image mode=L size=28x28 at 0x126895B00>, 5)
>>> fig = plt.figure(figsize=(15, 6))
>> for i, (image, label) in islice(enumerate(mnist_dataset), 10):
...        ax = fig.add_subplot(2, 5, i+1)
...        ax.set_xticks([]); ax.set_yticks([])
...        ax.imshow(image, cmap='gray_r')
...        ax.set_title(f'{label}', size=15)
>> plt.show()
```

Bu veri kümesinden alınan örnek el yazısı rakamlar aşağıdaki gibi gösterilmiştir:



Şekil 12.6: El yazısı rakamların doğru tanımlanması

Bu, veri kümeleri oluşturma ve işleme ile veri kümelerini
torchvision.datasets kütüphanesi. Daha sonra, PyTorch'ta NN modellerinin nasıl oluşturulacağını göreceğiz.

PyTorch'ta bir NN modeli oluşturma

Bu bölümde şimdiye kadar PyTorch'un tensörleri manipüle etmek ve verileri eğitim sırasında üzerinde yineleyebileceğimiz biçimlerde düzenlemek için temel yardımcı bileşenlerini öğrendiniz. Bu bölümde, nihayet PyTorch'ta ilk tahmin modelimizi uygulayacağız. PyTorch, scikit-learn gibi makine öğrenimi kütüphanelerinden biraz daha esnek ama aynı zamanda daha karmaşık olduğundan, basit bir doğrusal regresyon modeli ile başlayacağız.

 olimpikbarbell@gmail.com

PyTorch sinir ağı modülü (torch.nn)

torch.nn, NN'lerin oluşturulmasına ve eğitilmesine yardımcı olmak için geliştirilmiş zarif tasarımlı bir modüldür. Sadece birkaç satır kodla kolay pro-tipleme ve karmaşık modellerin oluşturulmasına olanak tanır.

Modülün gücünü tam olarak kullanmak ve probleminize göre özelleştirmek için ne yaptığını anlamanız gerekir. Bu anlayışı geliştirmek için, ilk olarak torch.nn modülünden herhangi bir özellik kullanmadan oyuncak bir veri kümesi üzerinde temel bir doğrusal regresyon modeli eğiteceğiz; temel PyTorch tensör işlemlerinden başka hiçbir şey kullanmayacağız.

Ardından, torch.nn ve torch.optim'den aşamalı olarak özellikler ekleyeceğiz. Aşağıdaki alt bölümlerde göreceğiniz gibi, bu modüller bir NN modeli oluşturmayı son derece kolay hale getirmektedir. Ayrıca, önceki bölümde öğrendiğiniz Dataset ve DataLoader gibi PyTorch'ta desteklenen veri kümesi işlem hattı işlevlerinden de yararlanacağız. Bu kitapta, NN modelleri oluşturmak için torch.nn modülünü kullanacağız.

PyTorch'ta bir NN oluşturmak için en yaygın kullanılan yaklaşım, katmanların bir ağ oluşturmak için istiflenmesine izin veren nn.Module'dür. Bu bize ileri geçiş üzerinde daha fazla kontrol sağlar. nn.Module sınıfını kullanarak bir NN modeli oluşturmanın örneklerini göreceğiz.

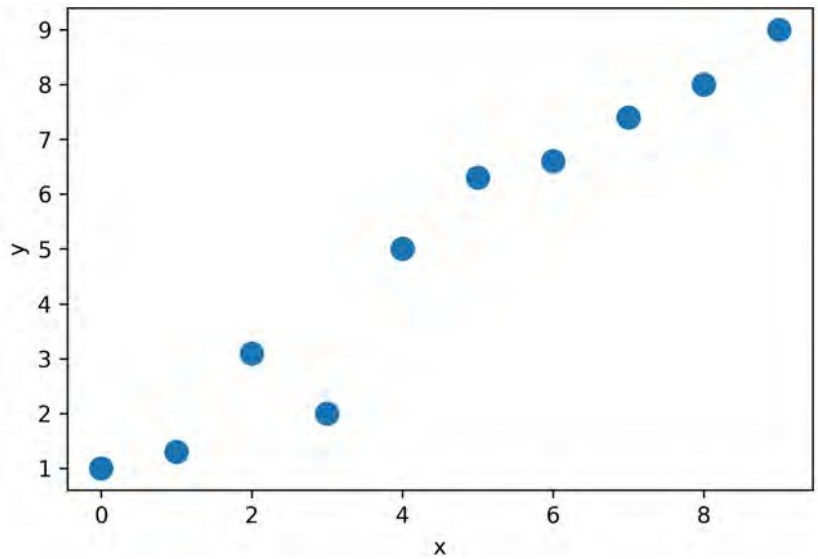
Son olarak, aşağıdaki alt bölümlerde göreceğiniz gibi, eğitilmiş bir model kaydedilebilir ve ileride kullanılmak üzere yeniden yüklenebilir.

Doğrusal regresyon modeli oluşturma

Bu alt bölümde, doğrusal bir regresyon problemini çözmek için basit bir model oluşturacağız. İlk olarak, NumPy'de bir oyuncak veri kümesi oluşturalım ve görselleştirelim:

```
>> X_train = np.arange(10, dtype='float32').reshape((10, 1))
>> y_train = np.array([1.0, 1.3, 3.1, 2.0, 5.0,
...                     6.3, 6.6,7.4, 8.0,
...                     9.0], dtype='float32')
>> plt.plot(X_train, y_train, 'o', markersize=10)
>> plt.xlabel('x')
>> plt.ylabel('y')
>> plt.show()
```

Sonuç olarak, eğitim örnekleri aşağıdaki gibi bir dağılım grafiğinde gösterilecektir:



Şekil 12.7: Eğitim örneklerinin bir dağılım grafiği

Ardından, özellikleri standartlaştıracağız (ortalama merkezleme ve standart sapmaya bölme) ve eğitim seti için bir PyTorch Veri Kümesi ve ilgili bir DataLoader oluşturacağız:

```
>> from torch.utils.data import TensorDataset
>>> X_train_norm = (X_train - np.mean(X_train)) / np.std(X_train)
>> X_train_norm = torch.from_numpy(X_train_norm)
>> y_train = torch.from_numpy(y_train).float()
>> train_ds = TensorDataset(X_train_norm, y_train)
>>> batch_size = 1
>> train_dl = DataLoader(train_ds, batch_size, shuffle=True)
```

Burada, DataLoader için 1'lik bir toplu iş boyutu ayarladık.

Şimdi, doğrusal regresyon için modelimizi $z = wx + b$ olarak tanımlayabiliriz. Burada torch.nn modülünü kullanacağız. Karmaşık NN modelleri oluşturmak için önceden tanımlanmış katmanlar sağlar, ancak başlangıç için sıfırdan bir modelin nasıl tanımlanacağını öğreneceksiniz. Bu bölümün ilerleyen kısımlarında, bu ön tanımlı katmanları nasıl kullanacağınızı göreceksiniz.

 olinpikbarbell@gmail.com

Bu regresyon problemi için sıfırdan bir doğrusal regresyon modeli tanımlayacağız. Modelimizin parametrelerini, sırasıyla ağırlık ve yanlılık parametrelerine karşılık gelen weight ve bias olarak tanımlayacağız. Son olarak, bu modelin çıktısını oluşturmak için girdi verilerini nasıl kullandığını belirlemek için model() işlevini tanımlayacağız:

```
>>> torch.manual_seed(1)
>>> ağırlık = torch.randn(1)
>> weight.requires_grad_()
>> bias = torch.zeros(1, requires_grad=True)
>> def model(xb):
...         xb @ weight + bias döndür
```

Modeli tanımladıktan sonra, optimum model ağırlıklarını bulmak için minimize etmek istediğimiz kayıp fonksiyonunu tanımlayabiliriz. Burada, kayıp fonksiyonumuz olarak **ortalama karesel hatayı** (**MSE**) seçeceğiz:

```
>> def loss_fn(input, target):
...         return (girdi-hedef).pow(2).mean()
```

Ayrıca, modelin ağırlık parametrelerini öğrenmek için stokastik gradyan inişi kullanacağız. Bu alt bölümde, bu eğitimi stokastik gradyan inişi prosedürü ile kendimiz uygulayacağız, ancak bir sonraki alt bölümde, aynı şeyi yapmak için optimizasyon paketi torch.optim'deki SGD yöntemini kullanacağız.

Stokastik gradyan iniş algoritmasını uygulamak için gradyanları hesaplamamız gerekir. Gradyanları manuel olarak hesaplamak yerine PyTorch'un torch.autograd.backward fonksiyonunu kullanacağız. Torch.autograd ve otomatik farklılaştırmayı uygulamak için farklı sınıflarını ve fonksiyonlarını *Bölüm 13*, *Going Deeper - The Mechanics of PyTorch*'da ele alacağız.

Şimdi, öğrenme oranını ayarlayabilir ve modeli 200 epok için eğitebiliriz. Modeli veri kümesinin toplu sürümüne karşı eğitmek için kullanılan kod aşağıdaki gibidir:

```
>> learning_rate = 0.001
>> num_epochs = 200
>> log_epochs = 10
>> for epoch in range(num_epochs):
...         for x_batch, y_batch in train_dl:
...             pred = model(x_batch)
...             kayıp = kayıp_fn(pred, y_batch.long())
...             loss.backward()
...         torch.no_grad() ile:
...             weight -= weight.grad * learning_rate
...             bias -= bias.grad * learning_rate
...             weight.grad.zero_()
...             bias.grad.zero_()
...         if epoch % log_epochs==0:
```

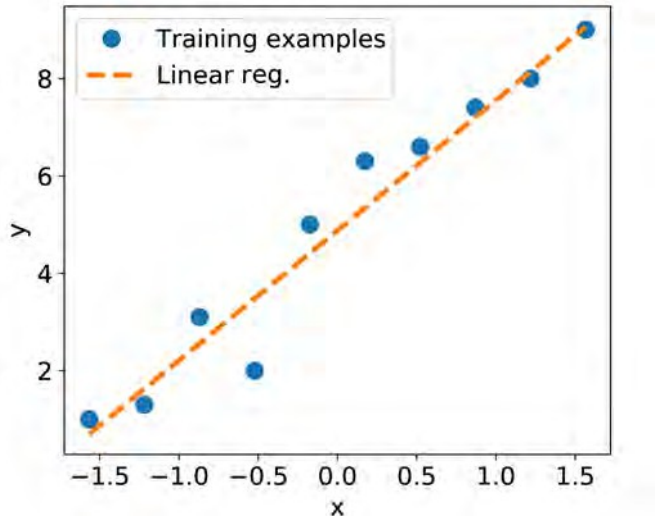
```
...             print(f'Epoch {epoch}  Kayıp {kayıp.item():.4f}')
Epoch  0   Kayıp 5.1701
Epoch  10   Kayıp 30.3370
Epoch  20   Kayıp 26.9436
Epoch  30   Kayıp 0,9315
Epoch  40   Kayıp 3.5942
Epoch  50   Kayıp 5,8960
Epoch  60   Kayıp 3.7567
Epoch  70   Kayıp 1,5877
Epoch  80   Kayıp 0,6213
Epoch  90   Kayıp 1.5596
Epoch  100   Kayıp 0,2583
Epoch  110   Kayıp 0,6957
Epoch  120   Kayıp 0,2659
Epoch  130   Kayıp 0,1615
Epoch  140   Kayıp 0,6025
Epoch  150   Kayıp 0,0639
Epoch  160   Kayıp 0,1177
Epoch  170   Kayıp 0,3501
Epoch  180   Kayıp 0,3281
Epoch  190   Kayıp 0,0970
```

Eğitilmiş modele bakalım ve grafiğini çizelim. Test verileri için, 0 ile 9 arasında eşit aralıklı değerlerden oluşan bir NumPy dizisi oluşturacağız. Modelimizi standartlaştırılmış özelliklerle eğittiğimiz için, aynı standartlaştırmayı test verilerine de uygulayacağız:

```
>> print('Final P a r a m e t e r s :', weight.item(),
bias.item()) Final Parameters: 2. 6 6 9 8 0 6 480407715
4.879569053649902
>> X_test = np.linspace(0, 9, num=100, dtype='float32').reshape(-1, 1)
>>> X_test_norm = (X_test - np.mean(X_train)) / np.std(X_train)
>> X_test_norm = torch.from_numpy(X_test_norm)
>> y_pred = model(X_test_norm).detach().numpy()
>>> fig = plt.figure(figsize=(13, 5))
>> ax = fig.add_subplot(1, 2, 1)
>> plt.plot(X_train_norm, y_train, 'o', markersize=10)
>> plt.plot(X_test_norm, y_pred, '--', lw=3)
>> plt.legend(['Eğitim örnekleri', 'Doğrusal reg.'], fontsize=15)
>> ax.set_xlabel('x', size=15)
>> ax.set_ylabel('y', size=15)
>> ax.tick_params(axis='both', which='major', labelsize=15)
>> plt.show()
```

 ilinpikbarbell@gmail.com

Şekil 12.8, eğitim örneklerinin ve eğitilmiş doğrusal regresyon modelinin bir dağılım grafiğini göstermektedir:



Şekil 12.8: Doğrusal regresyon modeli verilere iyi uyum sağlamaktadır

torch.nn ve torch.optim modülleri aracılığıyla model eğitimi

Önceki örnekte, özel bir kayıp fonksiyonu loss_fn() yazarak ve stokastik gradyan iniş optimizasyonu uygulayarak bir modelin nasıl eğitileceğini gördük. Ancak, kayıp fonksiyonunu ve gradyan güncellemelerini yazmak farklı projelerde tekrarlanabilir bir görev olabilir. torch.nn modülü bir dizi kayıp fonksiyonu sağlar ve torch.optim, hesaplanan gradyanlara dayalı olarak parametreleri güncellemek için çağrılabilen en yaygın kullanılan optimizasyon algoritmalarını destekler. Nasıl çalıştıklarını görmek için yeni bir MSE kayıp fonksiyonu ve stokastik gradyan iniş

```
>> import torch.nn as nn
>> kayıp_fn = nn.MSELoss(reduction='mean')
>>> input_size = 1
>>> output_size = 1
>>> model = nn.Linear(input_size, output_size)
>>>  optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

optimizeri oluşturalım:

Burada doğrusal katman için elle tanımlamak yerine torch.nn.Linear sınıfını kullandığımızı unutmayın.

Şimdi, modeli eğitmek için optimize edicinin step() yöntemini çağırabiliriz. Toplu bir veri kümesi aktarabiliriz (önceki örnekte oluşturulan train_dl gibi):

```
>> for epoch in range(num_epochs):
...     for x_batch, y_batch in train_dl:
...         # 1. Tahminler oluşturun
...         pred = model(x_batch)[:, 0]
...         # 2. Kaybı hesaplayın
...         kayıp = kayıp_fn(pred, y_batch)
...         # 3. Gradyanları hesaplayın
...         loss.backward()
...         # 4. Gradyanları kullanarak parametreleri güncelleyin
...         optimizer.step()
...         # 5. Gradyanları sıfıra sıfırlayın
...         optimizer.zero_grad()
...     if epoch % log_epochs==0:
...         print(f'Epoch {epoch}  Kayıp {kayıp.item():.4f}')
```

Model eğitildikten sonra sonuçları görselleştirin ve önceki yöntemin sonuçlarına benzer olduklarından emin olun. Ağırlık ve yanlılık parametrelerini elde etmek için aşağıdakileri yapabiliriz:

```
>> print('Nihai Parametreler:', model.weight.item(), model.bias.item())
Nihai Parametreler: 2.646660089492798 4.883835315704346
```

Iris veri setindeki çiçekleri sınıflandırmak için çok katmanlı bir algılayıcı oluşturma

Önceki örnekte, sıfırdan bir modelin nasıl oluşturulduğunu gördünüz. Bu modeli stokastik gradyan iniş optimizasyonu kullanarak eğittik. Yolculuğumuza mümkün olan en basit örneği temel alarak başlamış olsak da, bu kadar basit bir durum için bile modeli sıfırdan tanımlamanın ne çekici ne de iyi bir uygulama olduğunu görebilirsiniz. PyTorch bunun yerine torch.nn aracılığıyla bir NN modelinin yapı taşları olarak kolayca kullanılabilecek önceden tanımlanmış katmanlar sağlar. Bu bölümde, Iris çiçek veri kümesini kullanarak bir sınıflandırma görevini çözmek için bu katmanları nasıl kullanacağınızı (üç süsen türü arasında tanımlama) ve torch.nn modülünü kullanarak iki katmanlı bir perceptron oluşturmayı öğreneceksiniz. Öncelikle sklearn.datasets

```
>> from sklearn.datasets import load_iris
>> from sklearn.model_selection import train_test_split
>> iris = load_iris()
```

adresinden verileri alalım:

 alinpikbarbell@gmail.com

```
>> X = iris['veri']
>> y = iris['hedef']
>> X_train, X_test, y_train, y_test = train_test_split(
...        X, y, test_size=1./3, random_state=1)
```

Burada, eğitim için rastgele 100 örnek (2/3) ve test için 50 örnek (1/3) seçiyoruz.

Ardından, özellikleri standartlaştırıyoruz (ortalama merkezleme ve standart sapmaya bölme) ve eğitim seti için bir PyTorch Veri Kümesi ve ilgili bir DataLoader oluşturuyoruz:

```
>>> X_train_norm = (X_train - np.mean(X_train)) / np.std(X_train)
> >  X_train_norm = torch.from_numpy(X_train_norm).float()
>> y_train = torch.from_numpy(y_train)
>> train_ds = TensorDataset(X_train_norm, y_train)
>>> torch.manual_seed(1)
>>> batch_size = 2
>> train_dl = DataLoader(train_ds, batch_size, shuffle=True)
```

Burada, DataLoader için parti boyutunu 2 olarak ayarladık.

Şimdi, bir modeli verimli bir şekilde oluşturmak için torch.nn modülünü kullanmaya hazırız. Özellikle, nn.Module sınıfını kullanarak birkaç katmanı üst üste koyabilir ve bir NN oluşturabiliriz. Halihazırda mevcut olan tüm katmanların listesini https://pytorch.org/docs/stable/nn.html adresinde görebilirsiniz. Bu problem için, tam bağlantılı katman veya yoğun katman olarak da bilinen ve en iyi $f(w \times x + b)$ ile temsil edilebilen Doğrusal katmanı kullanacağız; burada x giriş özelliklerini içeren bir tensörü temsil eder, w ve b ağırlık matrisi ve önyargı vektörüdür ve f aktivasyon fonksiyonudur.

Bir NN'deki her katman, girdilerini bir önceki katmandan alır; bu nedenle, boyutluluğu (sıralama ve şekil) sabittir. Tipik olarak, yalnızca bir NN mimarisi tasarlarken çıktının boyutsallığı ile ilgilenmemiz gerekir. Burada, iki gizli katmanı olan bir model tanımlamak istiyoruz. Birincisi dört özellikten oluşan bir girdi alır ve bunları 16 nörona yansıtır. İkinci katman bir önceki katmanın (16 boyutlu) çıktısını alır ve üç sınıf etiketimiz olduğu için bunları üç çıkış nöronuna yansıtır. Bu şu şekilde yapılabilir:

```
>> class Model(nn.Module):
...        def init (self, input_size, hidden_size, output_size):
...            super(). init ()
...            self.layer1 = nn.Linear(input_size, h i d d e n _ s i z e )
...            s e l f . layer2 = nn.Linear(hidden_size, output_size)
...        def forward(self, x ):
...            x = self.layer1(x)
...            x = nn.Sigmoid()(x) x
...            = self.layer2(x)
...            x'i döndür
>>> input_size = X_train_norm.shape[1]
```

```
>> hidden_size = 16
>> output_size = 3
>>> model = Model(input_size, hidden_size, output_size)
```

Burada, ilk katman için sigmoid aktivasyon fonksiyonunu ve son (çıktı) katman için softmax aktivasyonunu kullandık. Son katmandaki softmax aktivasyonu çok sınıflı sınıflandırmayı desteklemek için kullanılır çünkü burada üç sınıf etiketimiz vardır (bu yüzden çıktı katmanında üç nöronumuz vardır). Farklı aktivasyon fonksiyonlarını ve bunların uygulamalarını bu bölümün ilerleyen kısımlarında tartışacağız.

Ardından, kayıp fonksiyonunu çapraz entropi kaybı ve optimize ediciyi Adam olarak belirliyoruz:

Adam optimizer, *14. Bölüm* olan *Derin Evrişimli Sinir Ağları ile Görüntüleri Sınıflandırma* bölümünde ayrıntılı olarak bahsedeceğimiz sağlam, gradyan tabanlı bir optimizasyon yöntemidir.

```
>> learning_rate = 0.001
>> kayıp_fn = nn.CrossEntropyLoss()
>>>  optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

Şimdi modeli eğitebiliriz. Epok sayısını 100 olarak belirleyeceğiz. Çiçek sınıflandırma modelini eğitme kodu aşağıdaki gibidir:

```
>> num_epochs = 100
>> kayıp_hist = [0] * num_epochs
>>> accuracy_hist = [0] * num_epochs
>> for epoch in range(num_epochs):
...         for x_batch, y_batch in train_dl:
...             pred = model(x_batch)
...             kayıp = kayıp_fn(pred, y_batch)
...             loss.backward()
...             optimizer.step()
...             optimizer.zero_grad()
...             loss_hist[epoch] += loss.item()*y_batch.size(0)
...             is_correct = (torch.argmax(pred, dim=1) == y_batch).float()
...             accuracy_hist[epoch] += is_correct.sum()
...         loss_hist[epoch] /= len(train_dl.dataset)
...         accuracy_hist[epoch] /= len(train_dl.dataset)
```

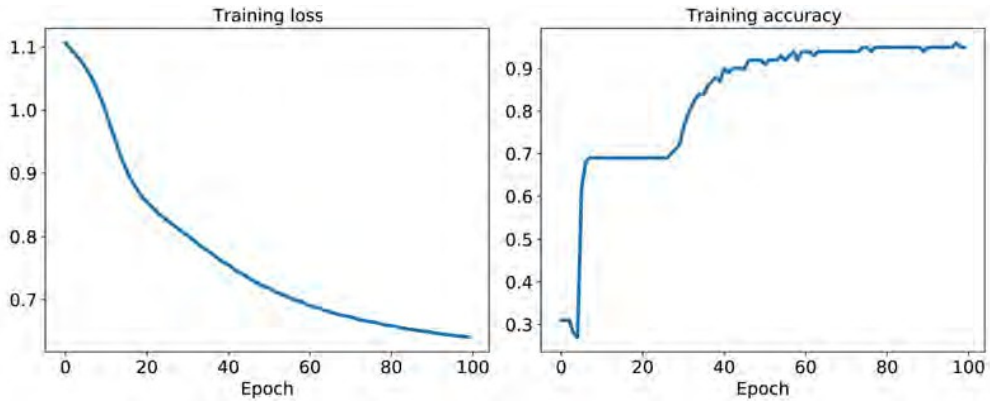
loss_hist ve accuracy_hist listeleri her epoktan sonra eğitim kaybını ve eğitim doğruluğunu tutar. Bunu öğrenme eğrilerini aşağıdaki gibi görselleştirmek için kullanabiliriz:

```
>>> fig = plt.figure(figsize=(12, 5))
>> ax = fig.add_subplot(1, 2, 1)
>>  ax.plot(loss_hist, lw=3)
```

alinpikbarbell@gmail.com

```
>> ax.set_title('Eğitim kaybı', size=15)
>> ax.set_xlabel('Epoch', size=15)
>> ax.tick_params(axis='both', which='major', labelsize=15)
>> ax = fig.add_subplot(1, 2, 2)
>> ax.plot(accuracy_hist, lw=3)
>> ax.set_title('Eğitim doğruluğu', size=15)
>> ax.set_xlabel('Epoch', size=15)
>> ax.tick_params(axis='both', which='major', labelsize=15)
>> plt.show()
```

Öğrenme eğrileri (eğitim kaybı ve eğitim doğruluğu) aşağıdaki gibidir:



Şekil 12.9: Eğitim kaybı ve doğruluk eğrileri

Eğitilen modelin test veri kümesi üzerinde değerlendirilmesi

Artık test veri kümesi üzerinde eğitilen modelin sınıflandırma doğruluğunu değerlendirebiliriz:

```
>>> X_test_norm = (X_test - np.mean(X_train)) / np.std(X_train)
>> X_test_norm = torch.from_numpy(X_test_norm).float()
>> y_test = torch.from_numpy(y_test)
>> pred_test = model(X_test_norm)
>>> doğru = (torch.argmax(pred_test, dim=1) == y_test).float()
>>> doğruluk = doğru.ortalama()
>> print(f'Test Acc.: {accuracy:.4f}') Test
Acc.: 0.9800
```

Modelimizi standartlaştırılmış özelliklerle eğittiğimiz için aynı standartlaştırmayı test verilerine de uyguladık. Sınıflandırma doğruluğu 0,98'dir (yani yüzde 98).

Eğitilmiş modeli kaydetme ve yeniden yükleme

Eğitilen modeller ileride kullanılmak üzere diske kaydedilebilir. Bu işlem aşağıdaki şekilde yapılabilir:

```
>>> path = 'iris_classifier.pt'
>> torch.save(model, path)
```

save(model) çağrısı hem model mimarisini hem de öğrenilen tüm parametreleri kaydedecektir. Genel bir kural olarak, modelleri 'pt' veya 'pth' dosya uzantısı kullanarak kaydedebiliriz.

Şimdi, kaydedilen modeli yeniden yükleyelim. Hem model mimarisini hem de ağırlıkları kaydettiğimiz için, parametreleri tek bir satırda kolayca yeniden oluşturabilir ve yeniden yükleyebiliriz:

```
>> model_new = torch.load(path)
```

model_new.eval() işlevini çağırarak model mimarisini doğrulamaya çalışın:

```
>> model_new.eval() Model(
   (katman1): Linear(in_features=4, out_features=16, bias=True)
   (layer2): Doğrusal(in_features=16, out_features=3, bias=True)
)
```

Son olarak, sonuçların öncekiyle aynı olduğunu doğrulamak için test veri kümesine yeniden yüklenen bu yeni modeli değerlendirelim:

```
>> pred_test = model_new(X_test_norm)
>>> doğru = (torch.argmax(pred_test, dim=1) == y_test).float()
>>> doğruluk = doğru.ortalama()
>> print(f'Test Acc.: {accuracy:.4f}') Test
Acc.: 0.9800
```

Yalnızca öğrenilen parametreleri kaydetmek istiyorsanız save(model.state_dict()) ifadesini aşağıdaki gibi kullanabilirsiniz:

```
>>> path = 'iris_classifier_state.pt'
>> torch.save(model.state_dict(), path)
```

Kaydedilen parametreleri yeniden yüklemek için, önce modeli daha önce yaptığımız gibi oluşturmamız, ardından yüklenen parametreleri modele beslememiz gerekir:

```
>> model_new = Model(input_size, hidden_size, output_size)
> >    model_new.load_state_dict(torch.load(path))
```

 alinpikbarbell@gmail.com

Çok katmanlı sinir ağları için aktivasyon fonksiyonlarının seçilmesi

Basitlik açısından, sigmoid aktivasyon fonksiyonunu şimdiye kadar sadece çok katmanlı ileri beslemeli NN'ler bağlamında ele aldık; *Bölüm 11*'deki MLP uygulamasında gizli katmanın yanı sıra çıkış katmanında da kullandık.

Bu kitapta sigmoidal lojistik fonksiyonun, $\sigma(z) = \frac{1}{1+e^{-z}}$, *sigmoid* fonksiyon olarak adlandırıldığını unutmayın Makine öğrenimi literatüründe yaygın olan kısalık için. Aşağıdaki alt bölümlerde, çok katmanlı NN'leri uygulamak için yararlı olan alternatif doğrusal olmayan işlevler hakkında daha fazla bilgi edineceksiniz.

Teknik olarak, farklılaştırılabilir olduğu sürece herhangi bir fonksiyonu çok katmanlı NN'lerde aktivasyon fonksiyonu olarak kullanabiliriz. Hatta Adaline'de olduğu gibi doğrusal aktivasyon fonksiyonları bile kullanabiliriz (*Bölüm 2*, *Sınıflandırma için Basit Makine Öğrenimi Algoritmalarının Eğitimi*). Ancak, pratikte, hem gizli hem de çıktı katmanları için doğrusal aktivasyon fonksiyonları kullanmak çok faydalı olmayacaktır, çünkü karmaşık problemlerin üstesinden gelebilmek için tipik bir yapay NN'ye doğrusal olmayan özellikler katmak istiyoruz. Doğrusal fonksiyonların toplamı sonuçta doğrusal bir fonksiyon verir.

Bölüm 11'de kullandığımız lojistik (sigmoid) aktivasyon fonksiyonu muhtemelen beyindeki bir nöron kavramını en yakından taklit eder - bunu bir nöronun ateşlenip ateşlenmeme olasılığı olarak düşünebiliriz. Bununla birlikte, lojistik (sigmoid) aktivasyon fonksiyonu, yüksek oranda negatif girdimiz varsa sorunlu olabilir, çünkü bu durumda sigmoid fonksiyonun çıktısı sıfıra yakın olacaktır. Sigmoid fonksiyonu sıfıra yakın bir çıktı verirse, NN çok yavaş öğrenecek ve eğitim sırasında kayıp manzarasının yerel minimumlarında sıkışıp kalma olasılığı daha yüksek olacaktır. Bu nedenle insanlar genellikle gizli katmanlarda aktivasyon fonksiyonu olarak hiperbolik tanjantı tercih ederler.

Hiperbolik tanjantın neye benzediğini tartışmadan önce, lojistik fonksiyonun bazı temellerini kısaca özetleyelim ve onu çok sınıflı sınıflandırma problemleri için daha kullanışlı hale getiren bir genellemeye bakalım.

Lojistik fonksiyon özeti

Bu bölümün giriş kısmında belirtildiği gibi, lojistik fonksiyon aslında sigmoid fonksiyonun özel bir durumudur. *Bölüm 3*, *Scikit-Learn Kullanarak Makine Öğrenimi Sınıflandırıcıları Turu'nda lojistik* regresyonla ilgili bölümden, ikili bir sınıflandırma görevinde x örneğinin pozitif sınıfa (sınıf 1) ait olma olasılığını modellemek için lojistik bir fonksiyon kullanabileceğimizi hatırlayacaksınız.

Verilen net girdi, z, aşağıdaki denklemde gösterilmiştir:

$$z = z_0 x_0 + z_1 x_1 + \cdots + z_m x_m = \sum_{i=0}^{m} z_i x_i = z^T x$$

Lojistik (sigmoid) fonksiyonu aşağıdakileri hesaplayacaktır:

$$\sigma_{logistic}(z) = \frac{1}{1 + e^{-z}}$$

w_0 'nin önyargı birimi olduğunu unutmayın (*y ekseni* kesişimi, yani $x_0 = 1$). Daha somut bir örnek vermek gerekirse, iki boyutlu bir veri noktası olan x için bir model ve w vektörüne atanmış aşağıdaki ağırlık katsayılarına sahip bir model ele alalım:

```
>> import numpy as np
>> X = np.array([1, 1.4, 2.5]) ## ilk değer 1 olmalıdır
>> w = np.array([0.4, 0 . 3, 0 . 5])
>> def net_input(X, w):
...         return np.dot(X, w)
>> def logistic(z):
...         return 1.0 / (1.0 + np.exp(-z))
>> def logistic_activation(X, w):
...         z = net_input(X, w)
...         return logistic(z)
>> print(f'P(y=1|x) = {logistic_activation(X, w):.3f}') P(y=1|x)
= 0,888
```

Net girdiyi (z) hesaplarsak ve bunu bu belirli özellik değerleri ve ağırlık katsayılarıyla bir lojistik nöronu etkinleştirmek için kullanırsak, 0,888 değerini elde ederiz; bu değeri, bu belirli örneğin (x) pozitif sınıfa ait olma olasılığının yüzde 88,8 olduğu şeklinde yorumlayabiliriz.

Bölüm 11'de, çok sınıflı temel gerçek etiketlerini temsil etmek için tek vuruşlu kodlama tekniğini kullandık ve çoklu lojistik aktivasyon birimlerinden oluşan çıktı katmanını tasarladık. Ancak, aşağıdaki kod örneğinde gösterileceği üzere, çoklu lojistik aktivasyon birimlerinden oluşan bir çıktı katmanı anlamlı, yorumlanabilir olasılık değerleri üretmez:

```
>> # W : shape = (n_output_units, n_hidden_units+1) ile dizi
>>> #       ilk sütunun önyargı birimleri olduğuna dikkat edin
>> W = np.array([[1.1, 1.2, 0.8, 0.4],
...                 [0.2, 0.4, 1.0, 0.2],
...                 [0.6, 1.5, 1.2, 0.7]])
>> # A : shape = (n_hidden_units + 1, n_samples) ile veri dizisi
>>> #       bu dizinin ilk sütununun 1 olması gerektiğine dikkat edin
>>> A = np.array([[1, 0.1, 0.4, 0.6]])
>>> Z = np.dot(W, A[0])
>> y_probas = logistic(Z)
>> print('Net Girdi: \n', Z)
Net Girdi:
[1.78 0.76 1.65]
>> print('Çıktı Birimleri:\n', y_probas)
Çıktı Birimleri:
[ 0.85569687 0 . 68135373 0 . 83889105]
```

 olimpikbarbell@gmail.com

Çıktıda da görebileceğiniz gibi, elde edilen değerler üç sınıflı bir problem için olasılık olarak yorumlanamaz. Ancak, modelimizi sınıf üyelik olasılıklarını değil de yalnızca sınıf etiketlerini tahmin etmek için kullanırsak, bu aslında büyük bir sorun değildir. Daha önce elde edilen çıktı birimlerinden sınıf etiketini tahmin etmenin bir yolu maksimum değeri kullanmaktır:

```
>> y_class = np.argmax(Z, axis=0)
>> print('Tahmin edilen sınıf etiketi:', y_class)
Tahmin edilen sınıf etiketi: 0
```

Bazı bağlamlarda, çok sınıflı tahminler için anlamlı sınıf olasılıkları hesaplamak faydalı olabilir. Bir sonraki bölümde, bu görevde bize yardımcı olabilecek lojistik fonksiyonun bir genellemesi olan softmax fonksiyonuna bir göz atacağız.

Softmax fonksiyonu ile çok sınıflı sınıflandırmada sınıf olasılıklarının tahmin edilmesi

Bir önceki bölümde argmax fonksiyonunu kullanarak nasıl sınıf etiketi elde edebileceğimizi gördünüz. Daha önce, *Iris veri* kümesindeki *çiçekleri sınıflandırmak için çok katmanlı bir algılayıcı oluşturma* bölümünde, MLP modelinin son katmanında aktivasyon='softmax' olarak belirledik. Softmax fonksiyonu argmax fonksiyonunun yumuşak bir formudur; tek bir sınıf indeksi vermek yerine her bir sınıfın olasılığını verir. Bu nedenle, çok sınıflı ortamlarda (multinomial lojistik regresyon) anlamlı sınıf olasılıkları hesaplamamıza olanak tanır.

Softmax'ta, z net girdisine sahip belirli bir örneğin birinci sınıfa ait olma olasılığı, paydada bir normalleştirme terimi, yani üstel olarak ağırlıklandırılmış doğrusal fonksiyonların toplamı ile hesaplanabilir:

$$p(z) = \sigma(z) = \frac{e^{z_i}}{\sum_{j=1}^{M} e^{z_j}}$$

Softmax'i çalışırken görmek için Python'da kodlayalım:

```
>> def softmax(z):
...     return np.exp(z) / np.sum(np.exp(z))
>> y_probas = softmax(Z)
>> print('Olasılıklar:\n', y_probas)
Olasılıklar:
[ 0.44668973 0 . 16107406 0 . 39223621]
>> np.sum(y_probas) 1.0
```

Gördüğünüz gibi, tahmin edilen sınıf olasılıkları artık beklediğimiz gibi 1'e toplanıyor. Tahmin edilen sınıf etiketinin, argmax fonksiyonunu lojistik çıktıya uyguladığımızdaki ile aynı olması da dikkat çekicidir.

Softmax fonksiyonunun sonucunu, çok sınıflı ortamlarda anlamlı sınıf üyeliği tahminleri elde etmek için yararlı olan *normalleştirilmiş* bir çıktı olarak düşünmek yardımcı olabilir. Bu nedenle, PyTorch'ta çok sınıflı bir sınıflandırma modeli oluşturduğumuzda, torch.softmax() işlevini bir girdi örnek grubu için her bir sınıf üyeliğinin olasılıklarını tahmin etmek için kullanabiliriz. PyTorch'ta torch.softmax() aktivasyon fonksiyonunu nasıl kullanabileceğimizi görmek için, aşağıdaki kodda Z'yi yığın boyutu için ayrılmış ek bir boyutla bir tensöre dönüştüreceğiz:

```
>> torch.softmax(torch.from_numpy(Z), dim=0) tensor([0.4467,
0 . 1611, 0 . 3922], dtype=torch.float64)
```

Hiperbolik tanjant kullanarak çıkış spektrumunu genişletme

Yapay NN'lerin gizli katmanlarında sıklıkla kullanılan bir diğer sigmoidal fonksiyon, lojistik fonksiyonun yeniden ölçeklendirilmiş bir versiyonu olarak yorumlanabilen **hiperbolik tanjanttır** (genellikle **tanh** olarak bilinir):

$$\sigma_{\text{logistic}}(z) = \frac{1}{1+e}$$

$$\sigma_{\tanh}(z) = 2 \times \sigma_{\text{logistic}}(2z) - 1 = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

Hiperbolik tanjantın lojistik fonksiyona göre avantajı, açık aralıkta (-1, 1) değişen daha geniş bir çıkış spektrumuna sahip olmasıdır, bu da geri yayılma algoritmasının yakınsamasını iyileştirebilir (*Neural Networks for Pattern Recognition, C. M. Bishop*, *Oxford University Press,* pages: 500-501, *1995*).

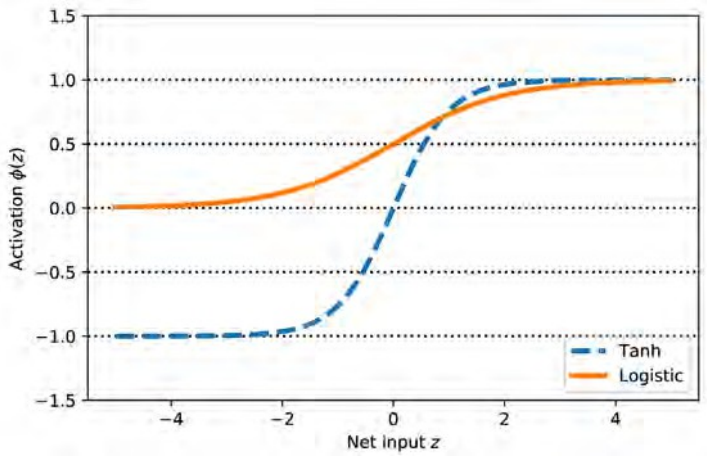
Buna karşılık, lojistik fonksiyon açık aralıkta (0, 1) değişen bir çıkış sinyali verir. Lojistik fonksiyon ile hiperbolik tanjantın basit bir karşılaştırması için iki sigmoidal fonksiyonu çizelim:

```
>> import matplotlib.pyplot as plt
>> def tanh(z):
...         e_p = np.exp(z)
...         e_m = np.exp(-z)
...         (e_p - e_m) / (e_p + e_m) döndürür
>> z = np.arange(-5, 5, 0.005)
>> log_act = logistic(z)
>> tanh_act = tanh(z)
>> plt.ylim([-1,5, 1,5])
>> plt.xlabel('net girdi $z$')
>> plt.ylabel('aktivasyon $\phi(z)$')
>> plt.axhline(1, color='black', linestyle=':')
>> plt.axhline(0.5, color='black', linestyle=':')
>> plt.axhline(0, color='black', linestyle=':')
```

 olimpikbarbell@gmail.com

```
>> plt.axhline(-0.5, color='black', linestyle=':')
>> plt.axhline(-1, color='black', linestyle=':')
>> plt.plot(z, tanh_act,
...          linewidth=3, linestyle='--',
...          label='tanh')
>> plt.plot(z, log_act,
...          çizgi genişliği=3,
...          label='logistic')
>> plt.legend(loc='sağ alt')
>> plt.tight_layout()
>> plt.show()
```

Gördüğünüz gibi, iki sigmoidal eğrinin şekilleri birbirine çok benziyor; ancak tanh fonksiyonu lojistik fonksiyonun iki katı çıktı alanına sahip:



Şekil 12.10: Tanh ve lojistik fonksiyonlarının karşılaştırılması

Daha önce lojistik ve tanh fonksiyonlarını örnekleme amacıyla ayrıntılı bir şekilde uyguladığımızı unutmayın. Pratikte NumPy'nin tanh fonksiyonunu kullanabiliriz.

Alternatif olarak, bir NN modeli oluştururken, aynı sonuçları elde etmek için PyTorch'ta torch.tanh(x) kullanabiliriz:

```
>> np.tanh(z)
array([-0.9999092 , -0.99990829, -0.99990737, ..., 0.99990644,
        0.99990737, 0 .99990829])
>>> torch.tanh(torch.from_numpy(z))
tensör([-0.9999, -0.9999, -0.9999, ..., 0.9999, 0.9999, 0.9999],
       dtype=torch.float64)
```

Buna ek olarak, lojistik fonksiyonu SciPy'nin özel modülünde mevcuttur:

```
>> from scipy.special import expit
>>> expit(z)
array([0.00669285, 0 . 00672617, 0.00675966, ..., 0.99320669, 0 . 99324034,
        0.99327383])
```

Benzer şekilde, aynı hesaplamayı yapmak için PyTorch'taki torch.sigmoid() işlevini aşağıdaki gibi kullanabiliriz:

```
>>> torch.sigmoid(torch.from_numpy(z))
tensör([0.0067, 0.0067, 0.0068, ..., 0.9932, 0.9932, 0.9933],
        dtype=torch.float64)
```

torch.sigmoid(x) kullanımının daha önce kullandığımız torch.nn.Sigmoid()(x) ile eşdeğer sonuçlar ürettiğine dikkat edin. torch.nn.Sigmoid, davranışı kontrol etmek amacıyla bir nesne oluşturmak için parametreler aktarabileceğiniz bir sınıftır. Buna karşılık, torch.sigmoid bir fonksiyondur.

Doğrultulmuş doğrusal birim aktivasyonu

Düzeltilmiş doğrusal birim (**ReLU**), derin NN'lerde sıklıkla kullanılan bir başka aktivasyon fonksiyonudur. ReLU'yu incelemeden önce, bir adım geriye gidip tanh ve lojistik aktivasyonların kaybolan gradyan sorununu anlamalıyız.

Bu sorunu anlamak için, başlangıçta $z_1 = 20$ net girdiye sahip olduğumuzu ve bunun $z_2 = 25$ olarak değiştiğini varsayalım. Tanh aktivasyonunu hesapladığımızda, $\sigma(z_1) = 1.0$ ve $\sigma(z_2) = 1.0$ elde ederiz, bu da çıktıda hiçbir değişiklik olmadığını gösterir (tanh fonksiyonunun asimptotik davranışı ve sayısal hatalar nedeniyle).

Bu, aktivasyonların net girdiye göre türevinin z büyüdükçe azaldığı anlamına gelir. Sonuç olarak, eğitim aşamasında ağırlıkların öğrenilmesi çok yavaş hale gelir çünkü türev terimleri sıfıra çok yakın olabilir. ReLU aktivasyonu bu sorunu çözmektedir. Matematiksel olarak ReLU aşağıdaki gibi tanımlanır:

$$\sigma(z) = \text{maks}(0, z)$$

ReLU hala NN'lerle karmaşık fonksiyonları öğrenmek için iyi olan doğrusal olmayan bir fonksiyondur. Bunun yanı sıra, ReLU'nun girdisine göre türevi, pozitif girdi değerleri için her zaman 1'dir. Bu nedenle, kaybolan gradyanlar sorununu çözerek derin NN'ler için uygun hale getirir. PyTorch'ta ReLU aktivasyonunu torch.relu() aşağıdaki gibi uygulayabiliriz:

```
>>> torch.relu(torch.from_numpy(z))
tensör([0.0000, 0.0000, 0.0000, ..., 4.9850, 4.9900, 4.9950],
        dtype=torch.float64)
```

ReLU aktivasyon fonksiyonunu bir sonraki bölümde çok katmanlı konvolüsyonel NN'ler için bir aktivasyon fonksiyonu olarak kullanacağız.

 alinpikbarbell@gmail.com

Artık yapay NN'lerde yaygın olarak kullanılan farklı aktivasyon fonksiyonları hakkında daha fazla bilgi sahibi olduğumuza göre, bu bölümü bu kitapta şimdiye kadar karşılaştığımız farklı aktivasyon fonksiyonlarına genel bir bakışla bitirelim:

Activation function	Equation	Example	1D graph
Linear	$\sigma(z) = z$	Adaline, linear regression	
Unit step (Heaviside function)	$\sigma(z) = \begin{cases} 0 & z < 0 \\ 0.5 & z = 0 \\ 1 & z > 0 \end{cases}$	Perceptron variant	
Sign (signum)	$\sigma(z) = \begin{cases} -1 & z < 0 \\ 0 & z = 0 \\ 1 & z > 0 \end{cases}$	Perceptron variant	
Piece-wise linear	$\sigma(z) = \begin{cases} 0 & z \leq -\frac{1}{2} \\ z + \frac{1}{2} & -\frac{1}{2} \leq z \leq \frac{1}{2} \\ 1 & z \geq \frac{1}{2} \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\sigma(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, multilayer NN	
Hyperbolic tangent (tanh)	$\sigma(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multilayer NN, RNNs	
ReLU	$\sigma(z) = \begin{cases} 0 & z < 0 \\ z & z > 0 \end{cases}$	Multilayer NN, CNNs	

Şekil 12.11: Bu kitapta ele alınan aktivasyon fonksiyonları

torch.nn modülünde bulunan tüm aktivasyon fonksiyonlarının listesini https://pytorch.org/docs/stable/nn.functional.html#non-linear-activation-functions adresinde bulabilirsiniz.

Özet

Bu bölümde, özellikle derin öğrenmeye odaklanarak sayısal hesaplamalar için açık kaynaklı bir kütüphane olan PyTorch'u nasıl kullanacağınızı öğrendiniz. PyTorch, GPU'ları desteklemek için ek karmaşıklığı nedeniyle NumPy'ye göre kullanımı daha zahmetli olsa da, büyük, çok katmanlı NN'leri çok verimli bir şekilde tanımlamamıza ve eğitmemize olanak tanır.

Ayrıca, karmaşık makine öğrenimi ve NN modelleri oluşturmak ve bunları verimli bir şekilde çalıştırmak için torch.nn modülünü kullanmayı öğrendiniz. Temel PyTorch tensör işlevselliği aracılığıyla sıfırdan bir model tanımlayarak PyTorch'ta model oluşturmayı keşfettik. Matris-vektör çarpımları düzeyinde programlama yapmamız ve her işlemin her ayrıntısını tanımlamamız gerektiğinde modelleri uygulamak sıkıcı olabilir. Ancak bunun avantajı, geliştiriciler olarak bu tür temel işlemleri birleştirmemize ve daha karmaşık modeller oluşturmamıza olanak sağlamasıdır. Daha sonra, NN modelleri oluşturmayı sıfırdan uygulamaktan çok daha kolay hale getiren torch.nn'yi keşfettik.

Son olarak, farklı aktivasyon fonksiyonlarını öğrendiniz ve bunların davranışlarını ve uygulamalarını anladınız. Özellikle, bu bölümde tanh, softmax ve ReLU'yu ele aldık.

Bir sonraki bölümde, yolculuğumuza devam edeceğiz ve PyTorch hesaplama grafikleri ve otomatik türev paketi ile çalışırken bulacağımız PyTorch'a daha derinlemesine dalacağız. Yol boyunca, gradyan hesaplamaları gibi birçok yeni kavram öğreneceksiniz.

Kitabımızın Discord alanına katılın

Benzer düşünen insanlarla tanışmak ve 2000'den fazla üyemizle birlikte öğrenmek için Discord topluluğumuza katılın:
https://packt.link/MLwPyTorch

6587191CA584E
olimpikbarbell@gmail.com

olimpikbarbell@gmail.com

13

Daha Derine Gitmek - PyTorch'un Mekaniği

Bölüm 12, *PyTorch ile Sinir Ağı Eğitimini Paralelleştirme'*de, tensörlerin nasıl tanımlanacağını ve manipüle edileceğini ele aldık ve girdi boru hatları oluşturmak için torch.utils.data modülü ile çalıştık. Ayrıca PyTorch sinir ağı modülünü (torch.nn) kullanarak Iris veri kümesini sınıflandırmak için çok katmanlı bir algılayıcı oluşturduk ve eğittik.

Artık PyTorch sinir ağı eğitimi ve makine öğrenimi ile ilgili bazı uygulamalı deneyimlerimiz olduğuna göre, PyTorch kütüphanesine daha derin bir dalış yapmanın ve gelecek bölümlerde daha gelişmiş derin öğrenme modelleri uygulamamıza olanak tanıyacak zengin özelliklerini keşfetmenin zamanı geldi.

Bu bölümde, NN'leri uygulamak için PyTorch'un API'sinin farklı yönlerini kullanacağız. Özellikle, standart mimarilerin uygulanmasını çok kolay hale getirmek için çoklu soyutlama katmanları sağlayan torch.nn modülünü tekrar kullanacağız. Ayrıca, daha fazla özelleştirme gerektiren araştırma odaklı projelerde çok yararlı olan özel NN katmanları uygulamamıza da olanak tanır. Bu bölümün ilerleyen kısımlarında böyle özel bir katman uygulayacağız.

torch.nn modülünü kullanarak model oluşturmanın farklı yollarını göstermek için klasik **özel veya** (**XOR**) problemini de ele alacağız. İlk olarak, Sequential sınıfını kullanarak çok katmanlı algılayıcılar oluşturacağız. Daha sonra, özel katmanlar tanımlamak için nn.Module alt sınıfı gibi diğer yöntemleri ele alacağız. Son olarak, ham girdiden tahmine kadar makine öğrenimi adımlarını kapsayan iki gerçek dünya projesi üzerinde çalışacağız.

Ele alacağımız konular aşağıdaki gibidir:

- PyTorch hesaplama grafiklerini anlama ve bunlarla çalışma
- PyTorch tensör nesneleri ile çalışma
- Klasik XOR problemini çözme ve model kapasitesini anlama
- PyTorch'un Sequential sınıfını ve nn.Module sınıfını kullanarak karmaşık NN modelleri oluşturma
- Otomatik farklılaştırma ve torch.autograd kullanarak gradyanları hesaplama

PyTorch'un temel özellikleri

Bir önceki bölümde, PyTorch'un bize makine öğrenimi algoritmalarını uygulamak ve çalıştırmak için ölçeklenebilir, çok platformlu bir programlama arayüzü sağladığını gördük. PyTorch, 2016'daki ilk sürümünden ve 2018'deki 1.0 sürümünden sonra, derin öğrenme için en popüler iki çerçeveden biri haline geldi. Statik muadillerine kıyasla daha esnek olma avantajına sahip olan dinamik hesaplama grafikleri kullanır. Dinamik hesaplama grafikleri hata ayıklama dostudur: PyTorch, grafik bildirimi ve grafik değerlendirme adımlarının birbirine eklenmesine izin verir. Tüm değişkenlere tam erişime sahipken kodu satır satır yürütebilirsiniz. Bu, NN'lerin geliştirilmesini ve eğitilmesini çok kolaylaştıran çok önemli bir özelliktir.

PyTorch açık kaynaklı bir kütüphane olmasına ve herkes tarafından ücretsiz olarak kullanılabilmesine rağmen, geliştirilmesi Facebook tarafından finanse edilmekte ve desteklenmektedir. Bu, kütüphaneyi sürekli olarak genişleten ve geliştiren büyük bir yazılım mühendisleri ekibini içerir. PyTorch açık kaynaklı bir kütüphane olduğu için, Facebook dışındaki diğer geliştiricilerin de güçlü desteğine sahiptir ve bu geliştiriciler hevesle katkıda bulunmakta ve kullanıcı geri bildirimi sağlamaktadır. Bu durum PyTorch kütüphanesini hem akademik araştırmacılar hem de geliştiriciler için daha kullanışlı hale getirmiştir. Bu faktörlerin bir başka sonucu da PyTorch'un yeni kullanıcılara yardımcı olmak için kapsamlı belgelere ve öğreticilere sahip olmasıdır.

PyTorch'un bir önceki bölümde de belirtilen bir diğer önemli özelliği, tekli veya çoklu **grafik işlem birimleriyle** (**GPU'lar**) çalışabilmesidir. Bu, kullanıcıların derin öğrenme modellerini büyük veri kümeleri ve büyük ölçekli sistemler üzerinde çok verimli bir şekilde eğitmelerine olanak tanır.

Son olarak, PyTorch mobil dağıtımı destekler, bu da onu üretim için çok uygun bir araç haline getirir.

Bir sonraki bölümde, PyTorch'ta bir tensör ve fonksiyonun bir hesaplama grafiği aracılığıyla nasıl birbirine bağlandığına bakacağız.

PyTorch'un hesaplama grafikleri

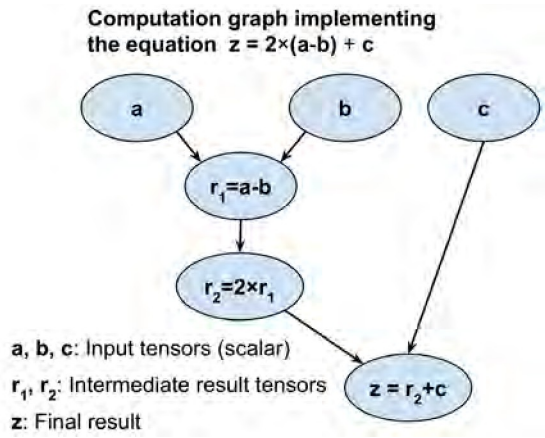
PyTorch hesaplamalarını **yönlendirilmiş asiklik grafiğe** (**DAG**) dayalı olarak gerçekleştirir. Bu bölümde, basit bir aritmetik hesaplama için bu grafların nasıl tanımlanabileceğini göreceğiz. Daha sonra, dinamik çizge paradigmasının yanı sıra PyTorch'ta çizgenin anında nasıl oluşturulduğunu göreceğiz.

Hesaplama grafiklerini anlama

PyTorch, özünde bir hesaplama grafiği oluşturmaya dayanır ve bu hesaplama grafiğini, girdiden çıktıya kadar tensörler arasındaki ilişkileri türetmek için kullanır. Diyelim ki a, b ve c rank 0 (skaler) tensörlerimiz var ve $z = 2 \times (a - b) + c$'yi değerlendirmek istiyoruz.

olimpikbarbell@gmail.com

Bu değerlendirme *Şekil 13.1*'de gösterildiği gibi bir hesaplama grafiği olarak temsil edilebilir:



Şekil 13.1: Bir hesaplama grafiği nasıl çalışır?

Gördüğünüz gibi, hesaplama grafiği basitçe düğümlerden oluşan bir ağdır. Her düğüm, giriş tensörüne veya tensörlerine bir işlev uygulayan ve çıktı olarak sıfır veya daha fazla tensör döndüren bir işlemi andırır. PyTorch bu hesaplama grafiğini oluşturur ve buna göre gradyanları hesaplamak için kullanır. Bir sonraki alt bölümde, PyTorch kullanarak bu hesaplama için bir grafik oluşturmanın bazı örneklerini göreceğiz.

PyTorch'ta bir grafik oluşturma

Önceki şekilde gösterildiği gibi $z = 2 \times (a - b) + c$'yi değerlendirmek için PyTorch'ta bir grafiğin nasıl oluşturulacağını gösteren basit bir örneğe bakalım. a, b ve c değişkenleri skalerdir (tek sayılar) ve bunları PyTorch tensörleri olarak tanımlarız. Grafiği oluşturmak için, örneğin giriş argümanları a, b ve c olan normal bir Python fonksiyonu tanımlayabiliriz:

```
>> import torch
>> def hesapla_z(a, b, c):
...     r1 = torch.sub(a, b)
...     r2 = torch.mul(r1, 2)
...     z = torch.add(r2, c)
...     dönüş z
```

Şimdi, hesaplamayı gerçekleştirmek için, bu fonksiyonu tensör nesneleriyle fonksiyon argümanları olarak çağırabiliriz. Add, sub (veya çıkarma) ve mul (veya çarpma) gibi PyTorch fonksiyonlarının, PyTorch tensör nesnesi biçiminde daha yüksek rütbeli girdiler sağlamamıza da izin verdiğini unutmayın. Aşağıdaki kod örneğinde, skaler girdilerin (rank 0) yanı sıra rank 1 ve

```
>> print('Skaler Girdiler:', compute_z(torch.tensor(1),
...          torch.tensor(2), torch.tensor(3))
Skaler Girdiler: tensör(1)
>> print('Sıra 1 Girdiler:', compute_z(torch.tensor([1]),
...          torch.tensor([2]), torch.tensor([3]))
Sıra 1 Girdiler: tensör([1])
>> print('Rank 2 Girdileri:', compute_z(torch.tensor([[1]])),
...          torch.tensor([[2]]), torch.tensor([[3]]))
Sıra 2 Girdiler: tensör([[1]])
```

rank 2 girdilerini liste olarak sağlıyoruz:

Bu bölümde, PyTorch'ta bir hesaplama grafiği oluşturmanın ne kadar basit olduğunu gördünüz. Daha sonra, model parametrelerini saklamak ve güncellemek için kullanılabilecek PyTorch tensörlerine bakacağız.

Model parametrelerini saklamak ve güncellemek için PyTorch tensör nesneleri

Tensör nesnelerini *Bölüm 12, PyTorch ile Sinir Ağı Eğitimini Paralelleştirme*'de ele almıştık. PyTorch'ta, gradyanların hesaplanması gereken özel bir tensör nesnesi, eğitim sırasında modellerimizin parametrelerini saklamamıza ve güncellememize olanak tanır. Böyle bir tensör, kullanıcı tarafından belirlenen başlangıç değerlerinde requires_grad değerinin True olarak atanmasıyla oluşturulabilir. Şu an itibariyle (2021 ortası), yalnızca kayan nokta ve karmaşık d tipi tensörlerin gradyan gerektirebileceğini unutmayın. Aşağıdaki kodda, float32 türünde tensör nesneleri oluşturacağız:

```
>> a = torch.tensor(3.14, requires_grad=True)
>> print(a)
tensor(3.1400, requires_grad=True)
>>> b = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
>> print(b)
tensor([1., 2., 3.], requires_grad=True)
```

 olimpikbarbell@gmail.com

requires_grad değerinin varsayılan olarak False olarak ayarlandığına dikkat edin. Bu değer requires_grad_() çalıştırılarak verimli bir şekilde True olarak ayarlanabilir.

method_() PyTorch'ta girdinin bir kopyasını oluşturmadan işlem yapmak için kullanılan bir in-place metodudur.

Aşağıdaki örneğe bir göz atalım:

```
>> w = torch.tensor([1.0, 2.0, 3.0])
>> print(w.requires_grad)
False
>> w.requires_grad_()
>> print(w.requires_grad)
True
```

NN modelleri için model parametrelerini rastgele ağırlıklarla başlatmanın geri yayılım sırasında simetriyi kırmak için gerekli olduğunu hatırlayacaksınız - aksi takdirde, çok katmanlı bir NN, lojistik regresyon gibi tek katmanlı bir NN'den daha kullanışlı olmazdı. Bir PyTorch tensörü oluştururken, rastgele bir başlatma şeması da kullanabiliriz. PyTorch, çeşitli olasılık dağılımlarına dayalı olarak rastgele sayılar üretebilir (bkz. https://pytorch.org/docs/stable/torch.html#random-sampling). Aşağıdaki örnekte, torch.nn.init modülünde de mevcut olan bazı standart başlatma yöntemlerine bir göz atacağız (bkz. https://pytorch.org/docs/stable/nn.init.html).

Şimdi, Xavier Glorot ve Yoshua Bengio tarafından önerilen klasik bir rastgele ilklendirme şeması olan Glorot ilklendirmesi ile nasıl bir tensör oluşturabileceğimize bakalım. Bunun için öncelikle GlorotNormal sınıfının bir nesnesi olarak boş bir tensör ve init adında bir operatör oluşturuyoruz. Daha sonra xavier_normal_() metodunu çağırarak bu tensörü Glorot ilklendirmesine göre değerlerle dolduruyoruz. Aşağıdaki örnekte, 2×3 şeklinde bir tensörü

```
>> import torch.nn as nn
>>> torch.manual_seed(1)
>> w = torch.empty(2, 3)
>>> nn.init.xavier_normal_(w)
>> print(w)
tensör([[ 0.4183, 0.1688, 0.0390],
        [ 0.3930, -0.2858, -0.1051]])
```

başlatıyoruz:

Xavier (veya Glorot) başlatma

Derin öğrenmenin erken gelişiminde, rastgele tekdüze veya rastgele normal ağırlık başlatmanın eğitim sırasında genellikle zayıf model performansıyla sonuçlanabileceği gözlemlenmiştir.

2010 yılında Glorot ve Bengio başlatmanın etkisini araştırmış ve derin ağların eğitimini kolaylaştırmak için yeni, daha sağlam bir başlatma şeması önermiştir. Xavier başlatmanın arkasındaki genel fikir, farklı katmanlar arasındaki gradyanların varyansını kabaca dengelemektir. Aksi takdirde, bazı katmanlar eğitim sırasında çok fazla dikkat çekerken diğer katmanlar geride kalabilir.

Glorot ve Bengio'nun araştırma makalesine göre, ağırlıkları tekdüze bir dağılımda başlatmak istiyorsak, bu tekdüze dağılımın aralığını aşağıdaki gibi seçmeliyiz:

$$WWW\ Uniform\ W\ \left(-\frac{\sqrt{6}}{\sqrt{nin + nout}}, \frac{\sqrt{6}}{\sqrt{nin + nout}}\right)$$

Burada $_{nin,}$ ağırlıklarla çarpılan girdi *nöronlarının* sayısıdır ve $_{nout,}$ bir sonraki katmana beslenen çıktı nöronlarının sayısıdır. Ağırlıkları Gauss (normal) dağılımından başlatmak için, bu Gauss'un standart sapmasını seçmenizi öneririz:

$$\sigma\ \sigma\ \frac{\sqrt{2}}{\sqrt{nin + nout}}$$

PyTorch, ağırlıkların hem tek tip hem de normal dağılımlarında Xavier başlatmayı destekler.

Glorot ve Bengio'nun başlatma şeması hakkında oranlama ve matematiksel motivasyon da dahil olmak üzere daha fazla bilgi için http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf adresinden ücretsiz olarak erişilebilen orijinal makaleyi (*Understanding the difficulty of deep feedforward neural networks*, *Xavier Glorot* ve *Yoshua Bengio*, 2010) tavsiye ederiz.

Şimdi, bunu daha pratik bir kullanım durumu bağlamına oturtmak için, iki Tensörü nasıl tanımlayabileceğimizi görelim
temel nn.Module sınıfı içindeki nesneler:

```
>> class MyModule(nn.Module):
...     def init (self):
...         super(). init ()
...         self.w1 = torch.empty(2, 3, requires_grad=True)
...         nn.init.xavier_normal_(self.w1)
...         self.w2 = torch.empty(1, 2, requires_grad=True)
...         nn.init.xavier_normal_(self.w2)
```

Bu iki tensör daha sonra gradyanları otomatik farklılaştırma yoluyla hesaplanacak olan ağırlıklar olarak kullanılabilir.

 olimpikbarbell@gmail.com

Otomatik türevlendirme yoluyla gradyanların hesaplanması

Bildiğiniz gibi, NN'leri optimize etmek, NN ağırlıklarına göre kaybın gradyanlarını hesaplamayı gerektirir. Bu, **stokastik gradyan inişi** (**SGD**) gibi optimizasyon algoritmaları için gereklidir. Buna ek olarak, gradyanların, bir NN modelinin bir test örneği için neden belirli bir tahminde bulunduğunu bulmak için ağı teşhis etmek gibi başka uygulamaları da vardır. Bu nedenle, bu bölümde bir hesaplamanın girdi değişkenlerine göre gradyanlarının nasıl hesaplanacağını ele alacağız.

Eğitilebilir değişkenlere göre kayıp gradyanlarının hesaplanması

PyTorch, iç içe geçmiş fonksiyonların gradyanlarını hesaplamak için *zincir kuralının* bir uygulaması olarak düşünülebilecek *otomatik türevi* destekler. Basitlik adına, *gradyan* terimini hem kısmi türevleri hem de gradyanları ifade etmek için kullanacağımızı unutmayın.

Kısmi türevler ve gradyanlar

Kısmi türev $\frac{\partial f}{\partial x_1}$ çok değişkenli bir fonksiyonun değişim oranı olarak anlaşılabilir. $f(x_1, x_2, \ldots)$ gibi birden fazla girdisi olan bir fonksiyonun, girdilerinden birine göre (burada: x_1). Bir fonksiyonun gradyanı, ∇f, tüm girdilerin kısmi türevlerinden oluşan bir vektördür. $\nabla f(x_1, x_2, \ldots) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots\right)$.

Bazı çıktılar ve hatta ara tensörlerle sonuçlanan bir dizi işlem tanımladığımızda PyTorch, hesaplanan bu tensörlerin hesaplama grafiğindeki bağımlı düğümlerine göre gradyanlarını hesaplamak için bir bağlam sağlar. Bu gradyanları hesaplamak için torch.autograd modülünden backward yöntemini çağırabiliriz. Bu yöntem, verilen tensörün gradyanlarının toplamını graftaki yaprak düğümlere (terminal düğümler) göre hesaplar.

$z = wx + b$ hesaplayacağımız ve kaybı hedef y ile tahmin z arasındaki karesel kayıp olarak tanımlayacağımız basit bir örnekle çalışalım, $Kayıp = (y - z)^2$. Birden fazla tahmin ve hedefe sahip olabileceğimiz daha genel durumda, kaybı karesel hataların toplamı olarak hesaplarız, $Loss\ L \sum_i (y_i - z)_i^2$. Bu hesaplamayı PyTorch'ta uygulamak için model parametreleri olan w ve b'yi tanımlayacağız, değişkenler (requires_gradient özelliği True olarak ayarlanmış tensörler) ve girdi, x ve y, varsayılan tensörler olarak. Kayıp tensörünü hesaplayacağız ve aşağıdaki gibi model parametrelerinin, w ve b, gradyanlarını hesaplamak için kullanacağız:

```
>> w = torch.tensor(1.0, requires_grad=True)
>>> b = torch.tensor(0.5, requires_grad=True)
>> x = torch.tensor([1.4])
>> y = torch.tensor([2.1])
>>> z = torch.add(torch.mul(w, x), b)
>> kayıp = (y-z).pow(2).sum()
```

```
>> kayıp.geriye doğru()
>> print('dL/dw : ', w.grad)
>> print('dL/db : ', b.grad)
dL/dw : tensör(-0,5600) dL/db
: tensör(-0,4000)
```

z değerini hesaplamak bir NN'de ileri bir geçiştir. $\frac{\partial Loss}{\partial w}$ ve $\frac{\partial Loss}{\partial b}$ değerlerini hesaplamak için kayıp tensörü üzerinde geriye doğru yöntemi kullandık. Bu çok basit bir örnek olduğundan, $\frac{\partial Loss}{\partial w} = 2x(wx + b - y)$ Hesaplanan gradyanların önceki kod örneğinde elde ettiğimiz sonuçlarla eşleştiğini doğrulamak için sembolik olarak:

```
>> # hesaplanan gradyanı doğrulama
>> print(2 * x * ((w * x + b) - y)) tensor([-
0.5600], grad_fn=<MulBackward0>)
```

b'nin doğrulanmasını okuyucuya bir alıştırma olarak bırakıyoruz.

Otomatik farklılaştırmayı anlama

Otomatik farklılaştırma, keyfi aritmetik işlemlerin gradyanlarını hesaplamak için bir dizi hesaplama tekniğini temsil eder. Bu işlem sırasında, bir hesaplamanın gradyanları (bir dizi işlem olarak ifade edilir), zincir kuralının tekrarlanan uygulamaları yoluyla gradyanların biriktirilmesiyle elde edilir. Otomatik türevin arkasındaki kavramı daha iyi anlamak için, $y = f(g(h(x)))$, girdi x ve çıktı y olan bir dizi iç içe hesaplama düşünelim:

- $u_0 = x$
- $u_1 = h(x)$
- $u_2 = g(u_1)$
- $u_3 = f(u_2) = y$

Türev $\frac{dy}{dx}$ iki farklı şekilde hesaplanabilir: ile başlayan ileri biriktirme $\frac{du3}{dx} = \frac{du3}{du2}\frac{du2}{du0}$, ve $\frac{dy}{du0} = \frac{dy}{du1}\frac{du1}{du0}$ ile başlayan ters biriktirme. PyTorch'un aşağıdakileri kullandığını unutmayın İkincisi, geriye yayılımı uygulamak için daha verimli olan ters biriktirme.

Çekişmeli örnekler

Girdi örneğine göre kaybın gradyanlarının hesaplanması, *karşıt örnekler* (veya *karşıt saldırılar*) üretmek için kullanılır. Bilgisayarla görmede düşman örnekler, giriş örneğine bazı küçük, algılanamayan gürültüler (veya pertürbasyonlar) eklenerek oluşturulan ve derin bir NN'nin bunları yanlış sınıflandırmasına neden olan örneklerdir. Muhalif örnekleri ele almak bu kitabın kapsamı dışındadır, ancak ilgileniyorsanız *Christian Szegedy ve diğerlerinin* orijinal makalesini bulabilirsiniz, *Intriguing properties of neural networks* https://arxiv.org/pdf/1312.6199.pdf.

 olimpikbarbell@gmail.com

torch.nn modülü aracılığıyla yaygın mimarilerin uygulamalarının basitleştirilmesi

İleri beslemeli bir NN modeli (örneğin, çok katmanlı bir algılayıcı) oluşturmanın ve nn.Module sınıfını kullanarak bir dizi katman tanımlamanın bazı örneklerini zaten görmüştünüz. nn.Module'ü daha derinlemesine incelemeden önce, nn.Sequential aracılığıyla bu katmanları bir araya getirmek için başka bir yaklaşıma kısaca göz atalım.

nn.Sequential tabanlı modellerin uygulanması

nn.Sequential (https://pytorch.org/docs/master/generated/torch.nn.Sequential.html#sequential) ile, model içinde depolanan katmanlar kademeli bir şekilde bağlanır. Aşağıdaki örnekte, iki yoğun (tam) bağlantılı katmana sahip bir model oluşturacağız:

```
>>> model = nn.Sequential(
...         nn.Linear(4, 16),
...         nn.ReLU(),
...         nn.Linear(16, 32),
...         nn.ReLU()
... )
>> model Sıralı(
  (0) : Linear(in_features=4, out_features=16, bias=True)
  (1) : ReLU()
  (2) : Linear(in_features=16, out_features=32, bias=True)
  (3) : ReLU()
)
```

Katmanları belirledik ve katmanları nn.Sequential sınıfına geçirdikten sonra modeli örnekledik. İlk tam bağlı katmanın çıktısı, ilk ReLU katmanının girdisi olarak kullanılır. İlk ReLU katmanının çıktısı, ikinci tam bağlı katman için girdi olur. Son olarak, ikinci tam bağlı katmanın çıktısı ikinci ReLU katmanının girdisi olarak kullanılır.

Bu katmanları, örneğin parametrelere farklı aktivasyon fonksiyonları, başlangıç izleyicileri veya düzenleme yöntemleri uygulayarak daha da yapılandırabiliriz. Bu kategorilerin çoğu için mevcut seçeneklerin kapsamlı ve eksiksiz bir listesi resmi belgelerde bulunabilir:

- Aktivasyon fonksiyonlarını seçme: https://pytorch.org/docs/stable/nn.html#non-linear- aktivasyon-ağırlıklı-toplam-doğrusal olmama
- Katman parametrelerinin nn.init aracılığıyla başlatılması: https://pytorch.org/docs/stable/nn.init. html
- Torch.optim'deki bazı optimize edicilerin weight_decay parametresi aracılığıyla katman parametrelerine (aşırı uyumu önlemek için) L2 düzenlemesi uygulamak: https://pytorch.org/docs/stable/optim. html

6587191CA584E

- Kayıp tensörüne L1 ceza terimini ekleyerek katman parametrelerine L1 düzenlemesi uygulamak (aşırı uyumu önlemek için), bunu daha sonra uygulayacağız

Aşağıdaki kod örneğinde, ağırlık için ilk değer dağılımını belirleyerek ilk tam bağlı katmanı yapılandıracağız. Ardından, ağırlık matrisi için L1 ceza terimini hesaplayarak ikinci tam bağlantılı katmanı yapılandıracağız:

```
> > >  nn.init.xavier_uniform_(model[0].weight)
>> l1_weight = 0.01
>> l1_ceza = l1_ağırlık * model[2].ağırlık.abs().sum()
```

Burada, ilk doğrusal katmanın ağırlığını Xavier başlatma ile başlattık. Ve ikinci doğrusal katmanın ağırlığının L1 normunu hesapladık.

Ayrıca, optimize edicinin türünü ve eğitim için kayıp fonksiyonunu da belirleyebiliriz. Yine, mevcut tüm seçeneklerin kapsamlı bir listesi resmi belgelerde bulunabilir:

- Torch.optim aracılığıyla optimize ediciler: https://pytorch.org/docs/stable/optim.html#algorithms
- Kayıp fonksiyonları: https://pytorch.org/docs/stable/nn.html#loss-functions

Bir kayıp fonksiyonu seçme

Optimizasyon algoritmaları için seçimlerle ilgili olarak, SGD ve Adam en yaygın kullanılan yöntemlerdir. Kayıp fonksiyonunun seçimi göreve bağlıdır; örneğin, bir regresyon problemi için ortalama kare hata kaybını kullanabilirsiniz.

Çapraz entropi kayıp fonksiyonları ailesi, *Derin Evrişimli Sinir Ağları ile Görüntülerin Sınıflandırılması* başlıklı *14. Bölümde* kapsamlı bir şekilde ele alınan sınıflandırma görevleri için olası seçenekleri sunar.

Ayrıca, önceki bölümlerde öğrendiğiniz teknikleri (*Bölüm 6*, *Model Değerlendirme ve Hiperparametre Ayarlama için En İyi Uygulamaları Öğrenme'd*eki model değerlendirme teknikleri gibi) problem için uygun ölçütlerle birlikte kullanabilirsiniz. Örneğin, hassasiyet ve geri çağırma, doğruluk, **eğri altındaki alan** (**AUC**) ve yanlış negatif ve yanlış pozitif puanlar sınıflandırma modellerini değerlendirmek için uygun ölçütlerdir.

Bu örnekte, ikili sınıflandırma için SGD iyileştiricisini ve çapraz entropi kaybını kullanacağız:

```
>> kayıp_fn = nn.BCELoss()
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
```

olimpikbarbell@gmail.com

Daha sonra, daha pratik bir örneğe bakacağız: klasik XOR sınıflandırma problemini çözme. İlk olarak, modeli oluşturmak için nn.Sequential() sınıfını kullanacağız. Yol boyunca, doğrusal olmayan karar sınırlarını ele almak için bir modelin kapasitesi hakkında da bilgi edineceksiniz. Ardından, bize ağın katmanları üzerinde daha fazla esneklik ve kontrol sağlayacak olan nn.Module aracılığıyla bir model oluşturmayı ele alacağız.

Bir XOR sınıflandırma problemini çözme

XOR sınıflandırma problemi, iki sınıf arasındaki doğrusal olmayan karar sınırını yakalamak açısından bir modelin kapasitesini analiz etmek için klasik bir problemdir. İki özelliği (x_0 , x_1) [-1, 1) arasında tekdüze bir dağılımdan alınan 200 eğitim örneğinden oluşan bir oyuncak veri kümesi oluşturuyoruz. Ardından, *i* eğitim örneği için temel gerçek etiketini aşağıdaki kurala göre atarız:

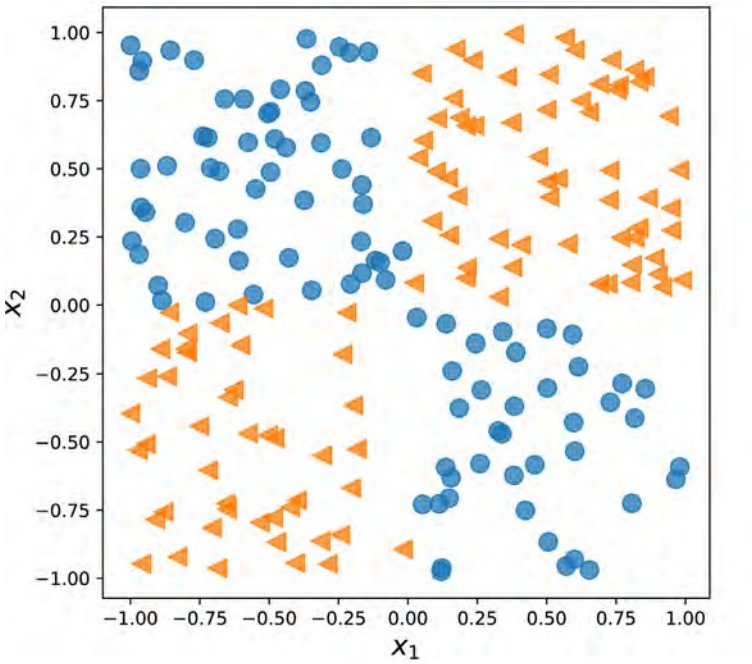
$$y^{(i)} = \begin{cases} x_0^{(i)} \times x_1^{(i)} < 0 \text{ ise } 0 \\ 1 \text{ aksi takdirde} \end{cases}$$

Verilerin yarısını (100 eğitim örneği) eğitim için, kalan yarısını da doğrulama için kullanacağız. Verileri oluşturmak ve eğitim ve doğrulama veri kümelerine ayırmak için kullanılan kod aşağıdaki

```
>> import matplotlib.pyplot as plt
>> import numpy as np
>>> torch.manual_seed(1)
>> np.random.seed(1)
>> x = np.random.uniform(low=-1, high=1, size=(200, 2))
>>> y = np.ones(len(x))
>>> y[x[:, 0] * x[:, 1]<0] = 0
>> n_train = 100
>> x_train = torch.tensor(x[:n_train, :], dtype=torch.float32)
>>  y_train = torch.tensor(y[:n_train], dtype=torch.float32)
>> x_valid = torch.tensor(x[n_train:, :], dtype=torch.float32)
>>  y_valid = torch.tensor(y[n_train:], dtype=torch.float32)
>>> fig = plt.figure(figsize=(6, 6))
>> plt.plot(x[y==0, 0], x[y==0, 1], 'o', alpha=0.75, markersize=10)
>> plt.plot(x[y==1, 0], x[y==1, 1], '<', alpha=0.75, markersize=10)
>>  plt.xlabel(r'$x_1$', size=15)
>>  plt.ylabel(r'$x_2$', size=15)
>> plt.show()
```

gibidir:

Kod, sınıf etiketlerine göre farklı işaretçilerle gösterilen eğitim ve doğrulama örneklerinin aşağıdaki dağılım grafiğiyle sonuçlanır:



Şekil 13.2: Eğitim ve doğrulama örneklerinin dağılım grafiği

Önceki alt bölümde, PyTorch'ta bir sınıflandırıcı uygulamak için ihtiyaç duyduğumuz temel araçları ele aldık. Şimdi bu görev ve veri kümesi için hangi mimariyi seçmemiz gerektiğine karar vermemiz gerekiyor. Genel bir kural olarak, ne kadar çok katmanımız ve her katmanda ne kadar çok nöronumuz varsa, modelin kapasitesi o kadar büyük olacaktır. Burada model kapasitesi, modelin karmaşık fonksiyonlara ne kadar kolay yaklaşabileceğinin bir ölçüsü olarak düşünülebilir. Daha fazla parametreye sahip olmak ağın daha karmaşık fonksiyonlara uyabileceği anlamına gelse de, daha büyük modellerin eğitilmesi genellikle daha zordur (ve aşırı uyuma eğilimlidir). Uygulamada, temel olarak basit bir modelle, örneğin lojistik regresyon gibi tek

```
>>> model = nn.Sequential(
...         nn.Linear(2, 1),
...         nn.Sigmoid()
... )
>> model Sıralı(
  (0): Linear(in_features=2, out_features=1, bias=True)
  (1): Sigmoid()
)
```

katmanlı bir NN ile başlamak her zaman iyi bir fikirdir:

 olimpikbarbell@gmail.com

Modeli tanımladıktan sonra, ikili sınıflandırma için çapraz entropi kayıp fonksiyonunu ve SGD optimizerini başlatacağız:

```
>> kayıp_fn = nn.BCELoss()
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
```

Ardından, tren verileri için 2'lik bir parti boyutu kullanan bir veri yükleyici oluşturacağız:

```
>> from torch.utils.data import DataLoader, TensorDataset
>> train_ds = TensorDataset(x_train, y_train)
>>> batch_size = 2
>>> torch.manual_seed(1)
>> train_dl = DataLoader(train_ds, batch_size, shuffle=True)
```

Şimdi modeli 200 epok için eğiteceğiz ve eğitim epoklarının bir geçmişini kaydedeceğiz:

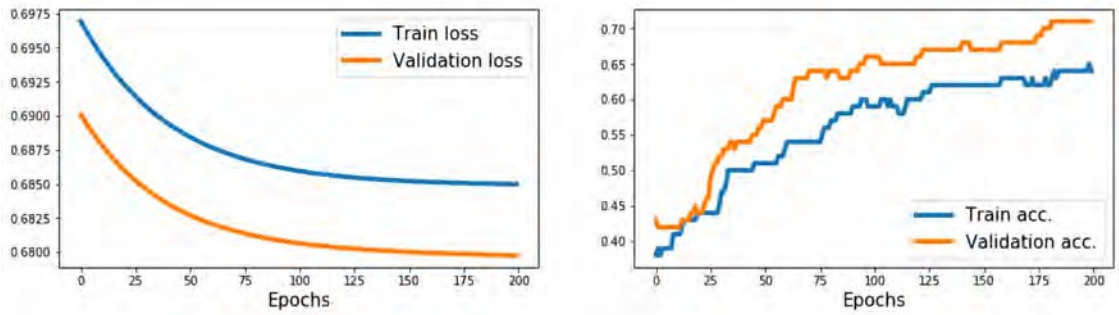
```
>>> torch.manual_seed(1)
>> num_epochs = 200
>> def train(model, num_epochs, train_dl, x_valid, y_valid):
...     loss_hist_train = [0] * num_epochs
...     accuracy_hist_train = [0] * num_epochs
...     loss_hist_valid = [0] * num_epochs
...     accuracy_hist_valid = [0] * num_epochs
...     for epoch in range(num_epochs):
...         for x_batch, y_batch in train_dl:
...             pred = model(x_batch)[:, 0]
...             kayıp = kayıp_fn(pred, y_batch)
...             loss.backward()
...             optimizer.step()
...             optimizer.zero_grad()
...             loss_hist_train[epoch] += loss.item()
...             is_correct = ((pred>=0.5).float() == y_batch).float()
...             accuracy_hist_train[epoch] += is_correct.mean()
...         loss_hist_train[epoch] /= n_train/batch_size
...         accuracy_hist_train[epoch] /= n_train/batch_size
...         pred = model(x_valid)[:, 0]
...         loss = loss_fn(pred, y_valid)
...         loss_hist_valid[epoch] = loss.item()
...         is_correct = ((pred>=0.5).float() == y_valid).float()
...         accuracy_hist_valid[epoch] += is_correct.mean()
...     return loss_hist_train, loss_hist_valid, \
...             accuracy_hist_train, accuracy_hist_valid
>>> history = train(model, num_epochs, train_dl, x_valid, y_valid)
```

Eğitim epoklarının geçmişinin, eğitimden sonra görsel inceleme için yararlı olan eğitim kaybı ve doğrulama kaybı ile eğitim doğruluğu ve doğrulama doğruluğunu içerdiğine dikkat edin. Aşağıdaki kodda, eğitim ve doğrulama kayıplarının yanı sıra doğrulukları da dahil olmak üzere öğrenme eğrilerini çizeceğiz.

Aşağıdaki kod eğitim performansını çizecektir:

```
>>> fig = plt.figure(figsize=(16, 4))
>> ax = fig.add_subplot(1, 2, 1)
>> plt.plot(history[0], lw=4)
>> plt.plot(history[1], lw=4)
>> plt.legend(['Tren kaybı', 'Doğrulama kaybı'], fontsize=15)
>> ax.set_xlabel('Epochs', size=15)
>> ax = fig.add_subplot(1, 2, 2)
>> plt.plot(history[2], lw=4)
>> plt.plot(history[3], lw=4)
>> plt.legend(['Train acc.', 'Validation acc.'], fontsize=15)
>> ax.set_xlabel('Epochs', size=15)
```

Bu, kayıplar ve doğruluklar için iki ayrı panel içeren aşağıdaki şekil ile sonuçlanır:



Şekil 13.3: Kayıp ve doğruluk sonuçları

Gördüğünüz gibi, gizli katmanı olmayan basit bir model yalnızca doğrusal bir karar sınırı türetebilir ve bu da XOR problemini çözemez. Sonuç olarak, hem eğitim hem de doğrulama veri kümeleri için kayıp terimlerinin çok yüksek olduğunu ve sınıflandırma doğruluğunun çok düşük olduğunu gözlemleyebiliriz.

Doğrusal olmayan bir karar sınırı türetmek için, doğrusal olmayan aktivasyon fonksiyonları ile bağlı bir veya daha fazla gizli katman ekleyebiliriz. Evrensel yaklaşım teoremi, tek bir gizli katmana ve nispeten çok sayıda gizli birime sahip ileri beslemeli bir NN'nin keyfi sürekli fonksiyonlara nispeten iyi yaklaşabileceğini belirtir. Bu nedenle, XOR problemini daha tatmin edici bir şekilde ele almak için bir yaklaşım, bir gizli katman eklemek ve doğrulama veri kümesinde tatmin edici sonuçlar gözlemleyene kadar farklı gizli birim sayılarını karşılaştırmaktır. Daha fazla gizli birim eklemek, bir katmanın genişliğini artırmaya karşılık gelecektir.

Alternatif olarak, modeli daha derin hale getirecek daha fazla gizli katman da ekleyebiliriz. Bir ağı daha geniş yapmak yerine daha derin yapmanın avantajı, karşılaştırılabilir bir model kapasitesi elde

olimpikbarbell@gmail.com

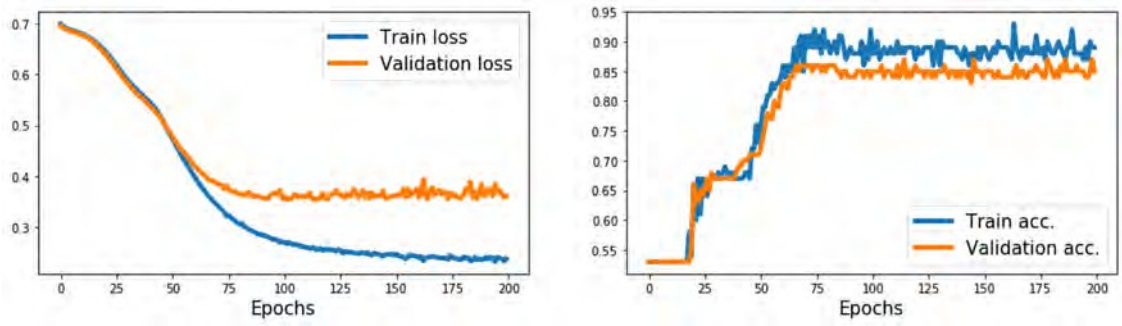
etmek için daha az parametreye ihtiyaç duyulmasıdır.

Bununla birlikte, derin (geniş modellere kıyasla) modellerin bir dezavantajı, derin modellerin kaybolma ve patlayan gradyanlara eğilimli olmasıdır, bu da onları eğitmeyi zorlaştırır.

Alıştırma olarak, her biri dört gizli birime sahip bir, iki, üç ve dört gizli katman eklemeyi deneyin. Aşağıdaki örnekte, iki gizli katmana sahip ileri beslemeli bir NN'nin sonuçlarına göz atacağız:

```
>>> model = nn.Sequential(
...         nn.Linear(2, 4),
...         nn.ReLU(),
...         nn.Linear(4, 4),
...         nn.ReLU(),
...         nn.Linear(4, 1),
...         nn.Sigmoid()
... )
>> kayıp_fn = nn.BCELoss()
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.015)
>> model Sıralı(
  (0): Linear(in_features=2, out_features=4, bias=True)
  (1): ReLU()
  (2): Linear(in_features=4, out_features=4, bias=True)
  (3): ReLU()
  (4): Linear(in_features=4, out_features=1, bias=True)
  (5): Sigmoid()
)
>>> history = train(model, num_epochs, train_dl, x_valid, y_valid)
```

Görselleştirme için önceki kodu tekrarlayabiliriz, bu da aşağıdakileri üretir:



Şekil 13.4: İki gizli katman ekleyen aker kayıp ve doğruluk sonuçları

Şimdi, modelin bu veriler için doğrusal olmayan bir karar sınırı türetebildiğini ve modelin eğitim veri kümesinde yüzde 100 doğruluğa ulaştığını görebiliriz. Doğrulama veri kümesinin doğruluğu yüzde 95'tir, bu da modelin biraz fazla uyum sağladığını gösterir.

olimpikbarbell@gmail.com

nn.Module ile model oluşturmayı daha esnek hale getirme

Önceki örnekte, birden fazla katmana sahip tam bağlantılı bir NN oluşturmak için PyTorch Sequential sınıfını kullandık. Bu, model oluşturmanın çok yaygın ve kullanışlı bir yoludur. Ancak, ne yazık ki birden fazla giriş, çıkış veya ara dalı olan daha karmaşık modeller oluşturmamıza izin vermiyor. İşte bu noktada nn.Module kullanışlı hale gelir.

Karmaşık modeller oluşturmanın alternatif yolu nn.Module'ü alt sınıflara ayırmaktır. Bu yaklaşımda, nn.Module'den türetilmiş yeni bir sınıf oluşturur ve init(_) yöntemini bir kurucu olarak tanımlarız. forward() yöntemi ileri geçişi belirtmek için kullanılır. init () yapıcı fonksiyonunda, katmanları sınıfın öznitelikleri olarak tanımlarız, böylece bunlara self reference özniteliği aracılığıyla erişilebilir. Ardından, forward() yönteminde, bu katmanların NN'nin ileri geçişinde nasıl kullanılacağını belirtiriz. Önceki modeli uygulayan yeni bir sınıf tanımlama kodu aşağıdaki

```
>> class MyModule(nn.Module):
...     def init (self):
...         super(). init () l1 =
...         nn.Linear(2, 4) a1 =
...         nn.ReLU()
...         l2 = nn.Lineer(4, 4)
...         a2 = nn.ReLU()
...         l3 = nn.Doğrusal(4,
...         1) a3 = nn.Sigmoid()
...         l = [l1, a1, l2, a2, l3, a3]
...         self.module_list = nn.ModuleList(l)
...
...     def ileri(self, x):
...         for f in self.module_list:
...             x = f(x)
...         x'i döndür
```

gibidir:

Tüm katmanları nn.ModuleList nesnesine koyduğumuza dikkat edin, bu nesne sadece aşağıdakilerden oluşan bir liste nesnesidir
nn.Module öğeleri. Bu, kodu daha okunabilir ve takip edilmesi daha kolay hale getirir.

Bu yeni sınıfın bir örneğini tanımladıktan sonra, daha önce yaptığımız gibi onu eğitebiliriz:

```
>> model = MyModule()
>> model
MyModule(
  (module_list): ModuleList(
    (0): Linear(in_features=2, out_features=4, bias=True)
    (1): ReLU()
    (2): Linear(in_features=4, out_features=4, bias=True)
    (3): ReLU()
```

```
      (4): Linear(in_features=4, out_features=1, bias=True)
      (5): Sigmoid()
    )
  )
>> kayıp_fn = nn.BCELoss()
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.015)
>>> history = train(model, num_epochs, train_dl, x_valid, y_valid)
```

Daha sonra, tren geçmişinin yanı sıra, doğrulama verilerini ve karar sınırını görselleştirmek için mlxtend kütüphanesini kullanacağız.

Mlxtend conda veya pip aracılığıyla aşağıdaki şekilde kurulabilir:

```
conda install mlxtend -c conda-forge
pip install mlxtend
```

Modelimizin karar sınırını hesaplamak için MyModule'e bir predict() yöntemi eklememiz gerekir Sınıf:

```
>>>     def predict(self, x):
...         x = torch.tensor(x, dtype=torch.float32)
...         pred = self.forward(x)[:, 0]
...         return (pred>=0.5).float()
```

Bir örnek için tahmin edilen sınıfı (0 veya 1) döndürür.

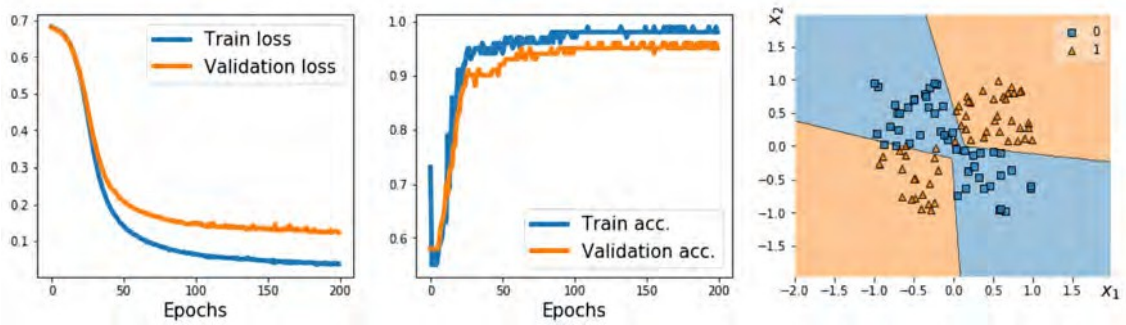
Aşağıdaki kod, eğitim performansını karar bölgesi yanlılığı ile birlikte çizecektir:

```
>> from mlxtend.plotting import plot_decision_regions
>>> fig = plt.figure(figsize=(16, 4))
>> ax = fig.add_subplot(1, 3, 1)
>> plt.plot(history[0], lw=4)
>> plt.plot(history[1], lw=4)
>> plt.legend(['Tren kaybı', 'Doğrulama kaybı'], fontsize=15)
>> ax.set_xlabel('Epochs', size=15)
>> ax = fig.add_subplot(1, 3, 2)
>> plt.plot(history[2], lw=4)
>> plt.plot(history[3], lw=4)
>> plt.legend(['Train acc.', 'Validation acc.'], fontsize=15)
>> ax.set_xlabel('Epochs', size=15)
>> ax = fig.add_subplot(1, 3, 3)
> > >   plot_decision_regions(X=x_valid.numpy(),
...                           y=y_valid.numpy().astype(np.integer),
...                           clf=model)
>>  ax.set_xlabel(r'$x_1$', size=15)
```

ilinpikbarbell@gmail.com

```
>> ax.xaxis.set_label_coords(1, -0.025)
>>  ax.set_ylabel(r'$x_2$', size=15)
>> ax.yaxis.set_label_coords(-0.025, 1)
>> plt.show()
```

Bu da kayıplar, doğruluklar ve doğrulama örneklerinin dağılım grafiği için üç ayrı panel ve karar sınırı ile birlikte *Şekil 13.5 ile* sonuçlanır:



Şekil 13.5: Bir dağılım grafiği de dahil olmak üzere sonuçlar

PyTorch'ta özel katmanlar yazma

PyTorch tarafından halihazırda desteklenmeyen yeni bir katman tanımlamak istediğimiz durumlarda, nn.Module sınıfından türetilmiş yeni bir sınıf tanımlayabiliriz. Bu özellikle yeni bir katman tasarlarken veya mevcut bir katmanı özelleştirirken kullanışlıdır.

Özel katmanların uygulanması kavramını açıklamak için basit bir örnek ele alalım. $ww(x + xx) + b$ hesaplayan yeni bir doğrusal katman tanımlamak istediğimizi düşünün; burada $\epsilon\epsilon$ gürültü değişkeni olarak rastgele bir değişkeni ifade eder. Bu hesaplamayı uygulamak için, nn.Module'ün bir alt sınıfı olarak yeni bir sınıf tanımlıyoruz. Bu yeni sınıf için hem yapıcı **init()** metodunu hem de forward() metodunu tanımlamamız gerekir. Yapıcıda, özelleştirilmiş katmanımız için değişkenleri ve diğer gerekli tensörleri tanımlarız. Yapıcıya input_size verilirse değişkenleri oluşturabilir ve yapıcıda başlatabiliriz. Alternatif olarak, değişken başlatmayı geciktirebilir (örneğin, tam girdi şeklini önceden bilmiyorsak) ve geç değişken oluşturma için bunu başka bir yönteme devredebiliriz.

Somut bir örneğe bakmak gerekirse, bir önceki paragrafta bahsedilen $ww(x + xx) + b$ hesaplamasını

```
>> class NoisyLinear(nn.Module):
...       def init (self, input_size, output_size,
...                        noise_stddev=0.1):
...            super(). init ()
...            w = torch.Tensor(input_size, output_size)
...            self.w = nn.Parameter(w) # nn.Parameter bir Tensördür
...                                     # bu bir modül parametresidir.
```

uygulayan NoisyLinear adında yeni bir katman tanımlayacağız:

```
...         nn.init.xavier_uniform_(self.w)
...         b = torch.Tensor(output_size).fill_(0)
...         self.b = nn.Parameter(b)
...         self.noise_stddev = noise_stddev
...
...     def forward(self, x, training=False):
...         eğer eğitimse:
...             noise = torch.normal(0.0, self.noise_stddev, x . shape)
...             x_new = torch.add(x, n o i s e )
...         Başka:
...             x_new = x
...         return torch.add(torch.mm(x_new, self.w), self.b)
```

Yapıcıya, Gauss dağılımından örneklenen $\epsilon\epsilon$ dağılımının standart sapmasını belirtmek için noise_stddev adında bir bağımsız değişken ekledik. Ayrıca, forward() yönteminde training=False adında ek bir bağımsız değişken kullandığımıza dikkat edin. Bunu, katmanın eğitim sırasında mı yoksa yalnızca tahmin (buna bazen *çıkarım* da denir) veya değerlendirme için mi kullanıldığını ayırt etmek için kullanıyoruz. Ayrıca, eğitim ve tahmin modlarında farklı davranan bazı yöntemler vardır. İlerleyen bölümlerde bu tür bir yöntem olan Dropout'un bir örneğiyle karşılaşacaksınız. Bir önceki kod parçasında, $\epsilon\epsilon$ rastgele vektörünün yalnızca eğitim sırasında oluşturulup girdiye ekleneceğini ve çıkarım veya değerlendirme için kullanılmayacağını da belirtmiştik.

Bir adım daha ileri gidip özel NoisyLinear katmanımızı bir modelde kullanmadan önce, basit bir örnek bağlamında test edelim.

1. Aşağıdaki kodda, bu katmanın yeni bir örneğini tanımlayacağız ve bir giriş tensörü üzerinde çalıştıracağız. Ardından, katmanı aynı giriş tensörü üzerinde üç kez çağıracağız:

```
>>> torch.manual_seed(1)
>> noisy_layer = NoisyLinear(4, 2)
>> x = torch.zeros((1, 4))
>> print(noisy_layer(x, training=True))
tensor([[ 0.1154, -0.0598]], grad_fn=<AddBackward0>)
>> print(noisy_layer(x, training=True))
tensor([[ 0.0432, -0.0375]], grad_fn=<AddBackward0>)
>> print(noisy_layer(x, training=False))
tensor([[0., 0.]], grad_fn=<AddBackward0>)
```

 ilinpikbarbell@gmail.com

İlk iki çağrının çıktılarının farklı olduğuna dikkat edin çünkü NoisyLinear katmanı giriş tensörüne rastgele gürültü eklemiştir. Üçüncü çağrı, training=False belirterek gürültü eklemediğimiz için [0, 0] çıktısı verir.

2. Şimdi, XOR sınıflandırma görevini çözmek için bir öncekine benzer yeni bir model oluşturalım. Daha önce olduğu gibi, model oluşturmak için nn.Module sınıfını kullanacağız, ancak bu sefer NoisyLinear katmanımızı çok katmanlı algılayıcının ilk gizli katmanı olarak kullanacağız.

```
>> class MyNoisyModule(nn.Module):
...     def init (self):
...         super(). init ()
...         self.l1 = NoisyLinear(2, 4, 0.07)
...         self.a1 = nn.ReLU()
...         self.l2 = nn.Lineer(4, 4)
...         self.a2 = n n . ReLU()
...         self.l3 = n n . Lineer(4, 1)
...         self.a3 = nn.Sigmoid()
...
...     def forward(self, x, training=False):
...         x = self.l1(x, eğitim) x =
...         self.a1(x)
...         x = self.l2(x) x
...         =  self.a2(x)  x
...         = self.l3(x) x =
...         self.a3(x)
...         return x
...
...     def predict(self, x):
...         x = torch.tensor(x, dtype=torch.float32)
...         pred = self.forward(x)[:, 0]
...         return (pred>=0.5).float()
...
>>> torch.manual_seed(1)
>>> model = MyNoisyModule()
>> model MyNoisyModule(
  (l1): NoisyLinear()
  (a1): ReLU()
  (l2): Linear(in_features=4, out_features=4, bias=True)
```

Kod aşağıdaki gibidir:

```
  (a2): ReLU()
  (l3): Linear(in_features=4, out_features=1, bias=True)
  (a3): Sigmoid()
)
```

3. Benzer şekilde, modeli daha önce yaptığımız gibi eğiteceğiz. Bu sefer, eğitim yığını üzerindeki tahmini hesaplamak için pred = model(x_batch)[:, 0] yerine pred = model(x_batch, True)[:, 0] kullanırız:

```
>> kayıp_fn = nn.BCELoss()
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.015)
>>> torch.manual_seed(1)
>> loss_hist_train = [0] * num_epochs
>>> accuracy_hist_train = [0] * num_epochs
>> loss_hist_valid = [0] * num_epochs
>>> accuracy_hist_valid = [0] * num_epochs
>> for epoch in range(num_epochs):
...     for x_batch, y_batch in train_dl:
...         pred = model(x_batch, True)[:, 0]
...         kayıp = kayıp_fn(pred, y_batch)
...         loss.backward()
...         optimizer.step()
...         optimizer.zero_grad()
...         loss_hist_train[epoch] += loss.item()
...         is_correct = (
...             (pred>=0.5).float() == y_batch
...         ).float()
...         accuracy_hist_train[epoch] += is_correct.mean()
...     loss_hist_train[epoch] /= n_train/batch_size
...     accuracy_hist_train[epoch] /= n_train/batch_size
...     pred = model(x_valid)[:, 0]
...     loss = loss_fn(pred, y_valid)
...     loss_hist_valid[epoch] = loss.item()
...     is_correct = ((pred>=0.5).float() == y_valid).float()
...     accuracy_hist_valid[epoch] += is_correct.mean()
```

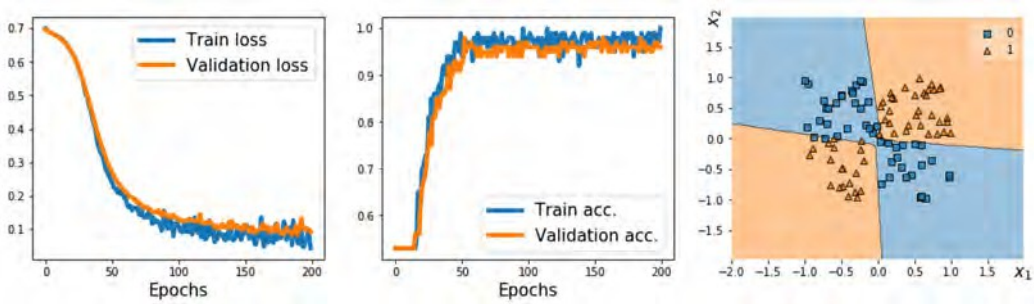
4. Model eğitildikten sonra, kayıpları, doğrulukları ve karar sınırını çizebiliriz:

```
>>> fig = plt.figure(figsize=(16, 4))
>> ax = fig.add_subplot(1, 3, 1)
>>  plt.plot(loss_hist_train, lw=4)
>>  plt.plot(loss_hist_valid, lw=4)
```

ilinpikbarbell@gmail.com

```
>> plt.legend(['Tren kaybı', 'Doğrulama kaybı'], fontsize=15)
>> ax.set_xlabel('Epochs', size=15)
>> ax = fig.add_subplot(1, 3, 2)
> >   plt.plot(accuracy_hist_train, lw=4)
> >   plt.plot(accuracy_hist_valid, lw=4)
>> plt.legend(['Train acc.', 'Validation acc.'], fontsize=15)
>> ax.set_xlabel('Epochs', size=15)
>> ax = fig.add_subplot(1, 3, 3)
>>> plot_decision_regions(
...         X=x_valid.numpy(),
...         y=y_valid.numpy().astype(np.integer),
...         clf=model
... )
>>  ax.set_xlabel(r'$x_1$', size=15)
>> ax.xaxis.set_label_coords(1, -0.025)
>>  ax.set_ylabel(r'$x_2$', size=15)
>> ax.yaxis.set_label_coords(-0.025, 1)
>> plt.show()
```

5. Ortaya çıkan şekil aşağıdaki gibi olacaktır:



Şekil 13.6: İlk gizli katman olarak NoisyLinear kullanılarak elde edilen sonuçlar

Burada amacımız, nn.Module alt sınıfından yeni bir özel katmanın nasıl tanımlanacağını öğrenmek ve bunu diğer standart torch.nn katmanlarını kullandığımız gibi kullanmaktı. Her ne kadar bu özel örnekte NoisyLinear performansı artırmaya yardımcı olmasa da, amacımızın esas olarak sıfırdan özelleştirilmiş bir katmanın nasıl yazılacağını öğrenmek olduğunu lütfen unutmayın. Genel olarak, yeni bir özelleştirilmiş katman yazmak, örneğin mevcut katmanların ötesinde yeni bir katmana bağlı olan yeni bir algoritma geliştirirseniz, diğer uygulamalarda yararlı olabilir.

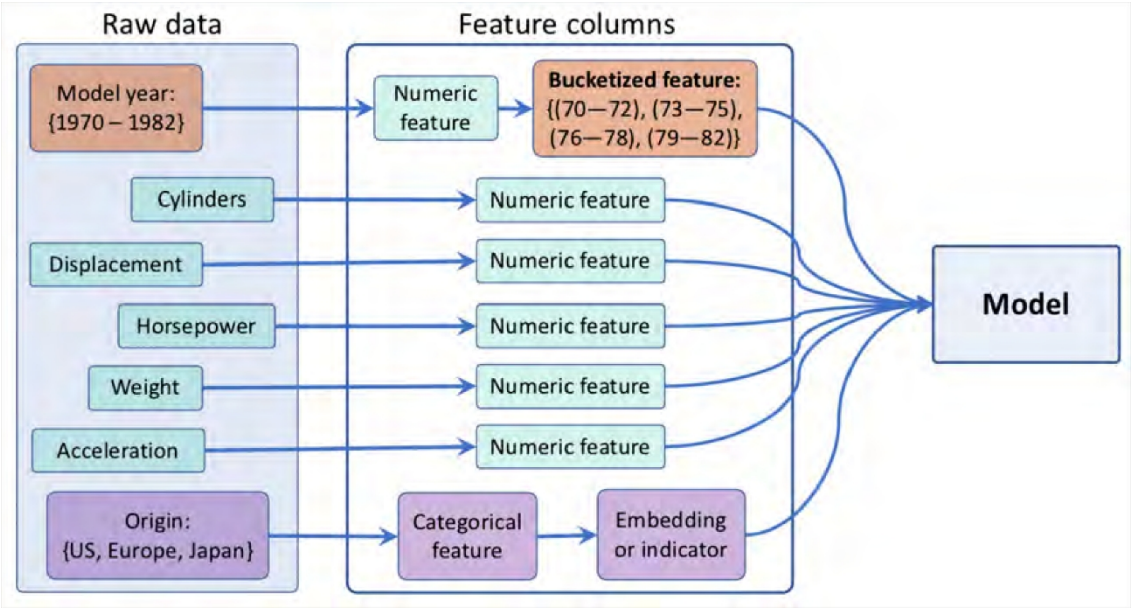
Birinci proje - bir otomobilin yakıt verimliliğinin tahmin edilmesi

Bu bölümde şimdiye kadar çoğunlukla torch.nn modülüne odaklandık. Basitlik açısından modelleri oluşturmak için nn.Sequential kullandık. Daha sonra, nn.Module ile model oluşturmayı daha esnek hale getirdik ve özelleştirilmiş katmanlar eklediğimiz ileri beslemeli NN'leri uyguladık. Bu bölümde, bir arabanın yakıt verimliliğini galon başına mil (MPG) cinsinden tahmin eden gerçek dünya projesi üzerinde çalışacağız. Veri ön işleme, özellik mühendisliği, eğitim, tahmin (çıkarım) ve değerlendirme gibi makine öğrenimi görevlerinin altında yatan adımları ele alacağız.

Özellik sütunları ile çalışma

Makine öğrenimi ve derin öğrenme uygulamalarında, çeşitli farklı özellik türleriyle karşılaşabiliriz: sürekli, sırasız kategorik (nominal) ve sıralı kategorik (ordinal). *Bölüm 4, İyi Eğitim Veri Kümeleri Oluşturma - Veri Önişleme'de* farklı özellik türlerini ele aldığımızı ve her bir türün nasıl *işleneceğini* öğrendiğimizi hatırlayacaksınız. Sayısal veriler sürekli veya ayrık olabilirken, PyTorch ile makine öğrenimi bağlamında "sayısal" verilerin özellikle kayan nokta türündeki sürekli verileri ifade ettiğini unutmayın.

Bazen özellik kümeleri farklı özellik türlerinin karışımından oluşur. Örneğin, *Şekil 13.7'de* gösterildiği gibi yedi farklı özellikten oluşan bir senaryo düşünün:



Şekil 13.7: Otomatik MPG veri yapısı

 olimpikbarbell@gmail.com

Şekilde gösterilen özellikler (model yılı, silindirler, yer değiştirme, beygir gücü, ağırlık, hızlanma ve menşe), bir otomobilin yakıt verimliliğini MPG cinsinden tahmin etmek için yaygın bir makine öğrenimi kıyaslama veri kümesi olan Auto MPG veri kümesinden elde edilmiştir. Veri setinin tamamına ve açıklamasına UCI'nin makine öğrenimi deposundan https://archive.ics.uci.edu/ml/datasets/auto+mpg adresinden ulaşılabilir.

Auto MPG veri setindeki beş özelliği (silindir sayısı, yer değiştirme, beygir gücü, ağırlık ve hızlanma) "sayısal" (burada sürekli) özellikler olarak ele alacağız. Model yılı, sıralı kategorik (ordinal) bir özellik olarak kabul edilebilir. Son olarak, üretim menşei, sırasıyla ABD, Avrupa ve Japonya'ya karşılık gelen 1, 2 ve 3 olmak üzere üç olası ayrık değere sahip sıralanmamış kategorik (nominal) bir özellik olarak kabul edilebilir.

Öncelikle verileri yükleyelim ve tamamlanmamış satırların çıkarılması, veri kümesinin eğitim ve test veri kümelerine bölünmesi ve sürekli özelliklerin standartlaştırılması dahil olmak üzere gerekli ön işleme adımlarını uygulayalım:

```
>> import pandas as pd
>> url = 'http://archive.ics.uci.edu/ml/' \
...          'machine-learning-databases/auto-mpg/auto-mpg.data'
>>> sütun_isimleri = ['MPG', 'Silindirler', 'Deplasman', 'Beygir Gücü',
...                   'Ağırlık', 'Hızlanma', 'Model Yılı', 'Menşei']
>> df = pd.read_csv(url, names=column_names,
...                  na_values = "?", comment='\t',
...                  sep=" ", skipinitialspace=True)
>>>
>> ## NA satırlarını bırakın
>> df = df.dropna()
>> df = d f .reset_index(drop=True)
>>>
>> ## eğitim/test bölünmeleri:
>> import sklearn
> >  import sklearn.model_selection
>> df_train, df_test = sklearn.model_selection.train_test_split(
...       df, train_size=0.8, random_state=1
... )
>>  train_stats = df_train.describe().transpose()
>>>
>> numeric_column_names = [
...       'Silindirler', 'Deplasman',
...       'Beygir Gücü', 'Ağırlık',
...       'İvme'
... ]
```

```
>> df_train_norm, df_test_norm = d f _ t r a i n .copy(), df_test.copy()
>> for col_name in numeric_column_names:
...         mean = train_stats.loc[col_name, 'mean']
...         std = train_stats.loc[col_name, 'std']
...         df_train_norm.loc[:, col_name] = \
...                 (df_train_norm.loc[:, col_name] - ortalama)/std
...         df_test_norm.loc[:, col_name] = \
...                 (df_test_norm.loc[:, col_name] - ortalama)/std
>> df_train_norm.tail()
```

Bu da aşağıdaki sonuçlara yol açar:

	MPG	Cylinders	Displacement	Horsepower	Weight	Acceleration	ModelYear	Origin
203	28.0	-0.824303	-0.901020	-0.736562	-0.950031	0.255202	76	3
255	19.4	0.351127	0.413800	-0.340982	0.293190	0.548737	78	1
72	13.0	1.526556	1.144256	0.713897	1.339617	-0.625403	72	1
235	30.5	-0.824303	-0.891280	-1.053025	-1.072585	0.475353	77	1
37	14.0	1.526556	1.563051	1.636916	1.470420	-1.359240	71	1

Şekil 13.8: Önceden işlenmiş Auto MG verileri

Önceki kod parçacığı aracılığıyla oluşturduğumuz pandas DataFrame, float türünde değerlere sahip beş sütun içerir. Bu sütunlar sürekli özellikleri oluşturacaktır.

Daha sonra, daha sonra eğiteceğimiz model için öğrenme görevini basitleştirmek amacıyla oldukça ince taneli model yılı (ModelYılı) bilgilerini kovalar halinde gruplayalım. Somut olarak, her bir aracı aşağıdaki gibi dört *yıl kümesinden* birine atayacağız:

$$\text{kova} = \begin{cases} 0 & \text{yıl} < 73 \text{ ise} \\ 1 & \text{if } 73 \leq \text{yıl} < 76 \\ 2 & 76 \leq \text{yıl} < 79 \text{ ise} \\ 3 & \text{yıl} \geq 79 \text{ ise} \end{cases}$$

Seçilen aralıkların "kova" kavramını göstermek için keyfi olarak seçildiğini unutmayın. Araçları bu kümeler halinde gruplandırmak için öncelikle üç kesme değeri tanımlayacağız: Model yılı özelliği için [73, 76, 79]. Bu kesme değerleri, örneğin (-∞, 73), [73, 76), [76, 79) ve [76, ∞) gibi yarı kapalı aralıkları belirtmek için kullanılır. Ardından, orijinal sayısal özellikler, kovaların indekslerini oluşturmak için torch.bucketize fonksiyonuna (https://pytorch.org/docs/stable/generated/torch.bucketize.html) aktarılacaktır. Kod aşağıdaki gibidir:

```
>>> sınırlar = torch.tensor([73, 76, 79])
>>>  v = torch.tensor(df_train_norm['Model Yılı'].values)
>> df_train_norm['Model Yılı Kovalandı'] = torch.bucketize(
```

 olimpikbarbell@gmail.com

```
...         v, boundaries, right=True
... )
>>> v = torch.tensor(df_test_norm['Model Yılı'].values)
>> df_test_norm['Kovalanan Model Yılı'] = torch.bucketize(
...         v, boundaries, right=True
... )
>> numeric_column_names.append('Model Yılı Kovalı')
```

Bu bucketized özellik sütununu Python listesi numeric_column_names'e ekledik.

Daha sonra, sıralanmamış kategorik özellik olan Origin için bir liste tanımlamaya devam edeceğiz. PyTorch'ta, kategorik bir özellikle çalışmanın iki yolu vardır: nn.Embedding (https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html) aracılığıyla bir gömme katmanı kullanmak veya tek vuruşlu kodlanmış vektörler (*gösterge olarak* da adlandırılır) kullanmak. Kodlama yaklaşımında, örneğin, 0 indeksi [1, 0, 0] olarak kodlanır, 1 indeksi [0, 1, 0] olarak kodlanır ve bu böyle devam eder. Öte yandan, gömme katmanı her bir indeksi float türünde rastgele sayılardan oluşan bir vektörle eşler ve bu vektör eğitilebilir. (Gömme katmanını, eğitilebilir bir ağırlık matrisiyle çarpılmış tek vuruşlu kodlamanın daha verimli bir uygulaması olarak düşünebilirsiniz).

Kategori sayısı fazla olduğunda, gömme katmanını kategori sayısından daha az boyutla kullanmak performansı artırabilir.

Aşağıdaki kod parçacığında, kategorik özelliği yoğun formata dönüştürmek için tek vuruşlu kodlama yaklaşımını kullanacağız:

```
>> from torch.nn.functional import one_hot
>>  toplam_orijin = len(set(df_train_norm['Origin']))
>> origin_encoded = one_hot(torch.from_numpy(
...         df_train_norm['Origin'].values) % total_origin)
>> x_train_numeric = torch.tensor(
...         df_train_norm[numeric_column_names].values)
>> x_train = torch.cat([x_train_numeric, origin_encoded], 1).float()
>> origin_encoded = one_hot(torch.from_numpy(
...         df_test_norm['Origin'].values) % total_origin)
>> x_test_numeric = torch.tensor(
...         df_test_norm[numeric_column_names].values)
>> x_test = torch.cat([x_test_numeric, origin_encoded], 1).float()
```

Kategorik özelliği üç boyutlu yoğun bir özelliğe kodladıktan sonra, bir önceki adımda işlediğimiz sayısal özelliklerle birleştirdik. Son olarak, temel gerçek MPG değerlerinden etiket tensörlerini aşağıdaki gibi oluşturacağız:

```
>>  y_train = torch.tensor(df_train_norm['MPG'].values).float()
>>>  y_test = torch.tensor(df_test_norm['MPG'].values).float()
```

Bu bölümde, PyTorch'ta ön işleme ve özellik oluşturma için en yaygın yaklaşımları ele aldık.

Bir DNN regresyon modelinin eğitimi

Şimdi, zorunlu özellikleri ve etiketleri oluşturduktan sonra, eğitilen veriler için 8'lik bir yığın boyutu kullanan bir veri yükleyici oluşturacağız:

```
>> train_ds = TensorDataset(x_train, y_train)
>>> batch_size = 8
>>> torch.manual_seed(1)
>> train_dl = DataLoader(train_ds, batch_size, shuffle=True)
```

Daha sonra, birinde 8, diğerinde 4 gizli birim bulunan iki tam bağlantılı katmana sahip bir model oluşturacağız:

```
>> hidden_units = [8, 4]
>>> input_size = x_train.shape[1]
>> all_layers = []
>> for hidden_unit in hidden_units:
...         katman = nn.Lineer(girdi_boyutu, gizli_birim)
...         all_layers.append(katman)
...         all_layers.append(nn.ReLU())
...         input_size = hidden_unit
>> all_layers.append(nn.Linear(hidden_units[-1], 1))
>>>  model = nn.Sequential(*all_layers)
>> model Sıralı(
  (0): Linear(in_features=9, out_features=8, bias=True)
  (1): ReLU()
  (2): Linear(in_features=8, out_features=4, bias=True)
  (3): ReLU()
  (4): Linear(in_features=4, out_features=1, bias=True)
)
```

Modeli tanımladıktan sonra, regresyon için MSE kayıp fonksiyonunu tanımlayacağız ve optimizasyon için stokastik gra- dient descent kullanacağız:

```
>> kayıp_fn = nn.MSELoss()
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
```

Şimdi modeli 200 epok için eğiteceğiz ve her 20 epok için tren kaybını göstereceğiz:

```
>>> torch.manual_seed(1)
>> num_epochs = 200
>> log_epochs = 20
```

ilinpikbarbell@gmail.com

```
>> for epoch in range(num_epochs):
...     loss_hist_train = 0
...     for x_batch, y_batch in train_dl:
...         pred = model(x_batch)[:, 0]
...         kayıp = kayıp_fn(pred, y_batch)
...         loss.backward()
...         optimizer.step()
...         optimizer.zero_grad()
...         loss_hist_train += loss.item()
...     if epoch % log_epochs==0:
...         print(f'Epoch {epoch} Kayıp '
...               f'{loss_hist_train/len(train_dl):.4f}')

Dönem 0 Kayıp 536.1047
Epoch  20   Kayıp 8.4361
Epoch  40   Kayıp 7.8695
Epoch  60   Kayıp 7.1891
Epoch  80   Kayıp 6.7062
Epoch  100   Kayıp 6.7599
Epoch  120   Kayıp 6.3124
Epoch  140   Kayıp 6.6864
Epoch  160   Kayıp 6.7648
Epoch  180   Kayıp 6.2156
```

200 epoktan sonra, eğitim kaybı 5 civarındaydı. Artık eğitilen modelin test veri kümesi üzerindeki regresyon performansını değerlendirebiliriz. Yeni veri noktalarındaki hedef değerleri tahmin etmek için özelliklerini modele besleyebiliriz:

```
>> torch.no_grad() ile:
...     pred = model(x_test.float())[:, 0]
...     loss = loss_fn(pred, y_test)
...     print(f'Test MSE: {loss.item():.4f}')
...     print(f'Test MAE: {nn.L1Loss()(pred, y_test).item():.4f}')
Test MSE: 9,6130
Test MAE: 2.1211
```

Test setindeki MSE 9.6 ve **ortalama mutlak hata** (**MAE**) 2.1'dir. Bu regresyon projesinden sonra, bir sonraki bölümde bir sınıflandırma projesi üzerinde çalışacağız.

Proje iki - MNIST el yazısı rakamlarının sınıflandırılması

Bu sınıflandırma projesi için MNIST el yazısı rakamlarını kategorize edeceğiz. Bir önceki bölümde, PyTorch'ta makine öğrenimi için dört temel adımı ayrıntılı olarak ele aldık ve bu bölümde tekrar etmemiz gerekecek.

Bölüm 12'de torchvision'dan mevcut veri kümelerini yükleme yöntemini öğrendiğinizi hatırlayacaksınız modülünü kullanacağız. İlk olarak, torchvision modülünü kullanarak MNIST veri kümesini yükleyeceğiz.

1. Kurulum adımı, veri setinin yüklenmesini ve hiperparametrelerin (eğitim seti ve test setinin boyutu ve mini partilerin boyutu) belirlenmesini içerir:

```
>> import torchvision
>> from torchvision import transforms
>> image_path = './'
>> transform = transforms.Compose([
...         transforms.ToTensor()
... ])
>> mnist_train_dataset = torchvision.datasets.MNIST(
...         root=image_path, train=True,
...         transform=transform, download=False
... )
>> mnist_test_dataset = torchvision.datasets.MNIST(
...         root=image_path, train=False,
...         transform=transform, download=False
... )
>>> batch_size = 64
>>> torch.manual_seed(1)
>> train_dl = DataLoader(mnist_train_dataset,
...                                 batch_size, shuffle=True)
```

   Burada, 64 örnekten oluşan bir veri yükleyici oluşturduk. Daha sonra, yüklenen veri kümelerini ön işleme tabi tutacağız.

2. Giriş özelliklerini ve etiketleri ön işleme tabi tutuyoruz. Bu projedeki özellikler, **Adım 1**'den okuduğumuz görüntülerin pikselleridir. torchvision. transforms.Compose kullanarak özel bir dönüşüm tanımladık. Bu basit durumda, dönüşümümüz yalnızca bir yöntemden oluşuyordu: ToTensor(). ToTensor() yöntemi piksel özelliklerini kayan tipte bir tensöre dönüştürür ve ayrıca pikselleri [0, 255] ila [0, 1] aralığında normalleştirir. *Derin Evrişimsel Sinir Ağları ile Görüntüleri Sınıflandırma* başlıklı *14. Bölümde,* daha karmaşık görüntü veri kümeleriyle çalışırken bazı ek veri dönüştürme yöntemleri göreceğiz. Etiketler, on basamağı temsil eden 0'dan 9'a kadar tam sayılardır. Dolayısıyla, herhangi bir ölçeklendirme veya başka bir dönüştürme yapmamıza gerek yoktur. Veri ö z e l l i ğ i n i kullanarak ham piksellere erişebileceğimizi unutmayın ve bunları [0, 1] aralığına ölçeklemeyi unutmayın.

   Veriler ön işleme tabi tutulduktan sonra bir sonraki adımda modeli oluşturacağız.

 ilinpikbarbell@gmail.com

3. NN modelini oluşturun:

```
>> hidden_units = [32, 16]
>>  image_size = mnist_train_dataset[0][0].shape
>>> input_size = image_size[0] * image_size[1] * image_size[2]
>> all_layers = [nn.Flatten()]
>> for hidden_unit in hidden_units:
...         katman = nn.Lineer(girdi_boyutu, gizli_birim)
...         all_layers.append(katman)
...         all_layers.append(nn.ReLU())
...         input_size = hidden_unit
>> all_layers.append(nn.Linear(hidden_units[-1], 10))
>>>  model = nn.Sequential(*all_layers)
>> model Sıralı(
  (0): Flatten(start_dim=1, end_dim=-1)
  (1): Linear(in_features=784, out_features=32, bias=True)
  (2): ReLU()
  (3): Linear(in_features=32, out_features=16, bias=True)
  (4): ReLU()
  (5): Linear(in_features=16, out_features=10, bias=True)
)
```

Modelin, bir giriş görüntüsünü tek boyutlu bir tensöre düzleştiren bir düzleştirme katmanı ile başladığını unutmayın. Bunun nedeni giriş görüntülerinin [1, 28, 28] şeklinde olmasıdır. Model, sırasıyla 32 ve 16 birimden oluşan iki gizli katmana sahiptir. Ve bir softmax fonksiyonu tarafından aktive edilen on sınıfı temsil eden on birimden oluşan bir çıktı katmanı ile sona erer. Bir sonraki adımda, modeli eğitim seti üzerinde eğiteceğiz ve test seti üzerinde değerlendireceğiz.

4. Modeli eğitim, değerlendirme ve tahmin için kullanın:

```
>> kayıp_fn = nn.CrossEntropyLoss()
>>> optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
>>> torch.manual_seed(1)
>> num_epochs = 20
>> for epoch in range(num_epochs):
...         accuracy_hist_train = 0
...         for x_batch, y_batch in train_dl:
```

```
...             pred = model(x_batch)
...             loss = loss_fn(pred, y_batch)
...             loss.backward()
...             optimizer.step()
...             optimizer.zero_grad()
...             is_correct = (
...                 torch.argmax(pred, dim=1) == y_batch
...             ).float()
...             accuracy_hist_train += is_correct.sum()
...         accuracy_hist_train /= len(train_dl.dataset)
...         print(f'Epoch {epoch} Doğruluk '
...               f'{accuracy_hist_train:.4f}')
Çağ 0       Doğruluk 0.8531
...
Dönem 9     Doğruluk 0.9691
...
Epoch 19 Doğruluk 0.9813
```

Çok sınıflı sınıflandırma için çapraz entropi kayıp fonksiyonunu ve gradyan inişi için Adam optimize edicisini kullandık. Adam optimizer hakkında *Bölüm 14*'te konuşacağız. Modeli 20 epok için eğittik ve her epok için eğitme doğruluğunu gösterdik. Eğitilen model eğitim setinde yüzde 96,3'lük bir doğruluğa ulaştı ve bunu test setinde değerlendireceğiz:

```
>>> pred = model(mnist_test_dataset.data / 255.)
>> is_correct = (
...         torch.argmax(pred, dim=1) ==
...         mnist_test_dataset.targets
... ).float()
>> print(f'Test doğruluğu: {is_correct.mean():.4f}')
Test doğruluğu: 0,9645
```

Test doğruluğu yüzde 95,6'dır. PyTorch kullanarak bir sınıflandırma problemini nasıl çözeceğinizi öğrendiniz.

Daha üst düzey PyTorch API'leri: PyTorch-Lightning'e kısa bir giriş

Son yıllarda PyTorch topluluğu Py- Torch'un üzerine birkaç farklı kütüphane ve API geliştirmiştir. Önemli örnekler arasında fastai (https://docs.fast.ai/), Catalyst (https://github.com/ catalyst-team/catalyst), PyTorch Lightning (https://www.pytorchlightning.ai), (https:// lightning-flash.readthedocs.io/en/latest/quickstart.html) ve PyTorch-Ignite (https:// github.com/pytorch/ignite) bulunmaktadır.

 olimpikbarbell@gmail.com

Bu bölümde PyTorch Lightning'i (kısaca Lightning) keşfedeceğiz, bu kütüphane yaygın olarak kullanılan bir Py- Torch kütüphanesidir ve derin sinir ağlarının eğitimini basitleştirir. Bununla birlikte, Lightning'in odak noktası basitlik ve esneklik olsa da, https://pytorch-lightning.rtfd.io/en/latest/ adresindeki resmi belgelerde öğrenebileceğiniz çoklu GPU desteği ve hızlı düşük hassasiyetli eğitim gibi birçok gelişmiş özelliği kullanmamıza da olanak tanır.

Ayrıca https://github.com/rasbt/ machine-learning-book/blob/main/ch13/ch13_part4_ignite.ipynb adresinde PyTorch-Ignite için bonus bir giriş bulunmaktadır.

Daha önceki bir bölümde, *Proje iki - MNIST el yazısı rakamlarının sınıflandırılması*, MNIST veri kümesindeki el yazısı rakamlarını sınıflandırmak için çok katmanlı bir algılayıcı uyguladık. Sonraki alt bölümlerde, bu sınıflandırıcıyı Lightning kullanarak yeniden uygulayacağız.

PyTorch Lightning'i Yükleme

Lightning, tercihinize bağlı olarak pip veya conda aracılığıyla kurulabilir. Örneğin Lightning'i pip ile kurmak için komut aşağıdaki gibidir:

```
pip install pytorch-lightning
```

Lightning'i conda aracılığıyla yüklemek için komut aşağıdadır:

```
conda install pytorch-lightning -c conda-forge
```

Aşağıdaki alt bölümlerdeki kod PyTorch Lightning 1.5 sürümüne dayanmaktadır, bu komutlarda pytorch-lightning yerine pytorch-lightning==1.5 yazarak yükleyebilirsiniz.

PyTorch Lightning modelini kurma

Sonraki alt bölümlerde eğiteceğimiz modeli uygulayarak başlıyoruz. Lightning için bir model tanımlamak, normal Python ve PyTorch koduna dayandığından nispeten basittir. Bir Lightning modelini uygulamak için gereken tek şey, normal PyTorch modülü yerine LightningModule kullanmaktır. PyTorch'un eğitmen API'si ve otomatik günlük kaydı gibi kolaylık sağlayan işlevlerinden yararlanmak için, aşağıdaki kodda göreceğimiz özel olarak adlandırılmış birkaç

```
import pytorch_lightning as pl
import torch
import torch.nn as nn

from torchmetrics import Doğruluk

sınıf MultiLayerPerceptron(pl.LightningModule):
    def init (self, image_shape=(1, 28, 28), hidden_units=(32, 16)):
        super(). init ()
```

yöntem tanımlamamız yeterlidir:

```
        # yeni PL nitelikleri:
        self.train_acc  =  Accuracy()
        self.valid_acc  =  Accuracy()
        self.test_acc                =
        A c c u r a c y ()

        # Önceki bölüme benzer model:
        input_size = image_shape[0] * image_shape[1] * image_shape[2]
        all_layers = [nn.Flatten()]
        for hidden_unit in hidden_units:
            layer = nn.Linear(input_size, hidden_unit)
            all_layers.append(layer)
            all_layers.append(nn.ReLU())
            input_size = hidden_unit

        all_layers.append(nn.Linear(hidden_units[-1], 10))
        self.model = nn.Sequential(*all_layers)

    def forward(self, x ): x
        =     self.model(x)
        return x

    def training_step(self, b a t c h , batch_idx):
        x, y = batch
        logits = self(x)
        kayıp = nn.functional.cross_entropy(self(x), y)
        preds = torch.argmax(logits, dim=1)
        self.train_acc.update(preds, y)
        self.log("train_loss", kayıp, prog_bar=True)
        return kayıp

    def training_epoch_end(self, outs):
        self.log("train_acc", s e l f .train_acc.compute())

    def validation_step(self, b a t c h , batch_idx):
        x, y = batch
        logits = self(x)
        kayıp = nn.functional.cross_entropy(self(x), y)
        preds = torch.argmax(logits, dim=1)
        self.valid_acc.update(preds, y)
        self.log("valid_loss", kayıp, prog_bar=True)
        self.log("valid_acc", self.valid_acc.compute(), prog_bar=True)
```

 olimpikbarbell@gmail.com

```
        geri dönüş kaybı

    def test_step(self, b a t c h ,
        batch_idx): x, y = batch
        logits = self(x)
        kayıp = nn.functional.cross_entropy(self(x), y)
        preds = torch.argmax(logits, dim=1)
        self.test_acc.update(preds, y)
        self.log("test_loss", kayıp, prog_bar=True)
        self.log("test_acc", self.test_acc.compute(), prog_bar=True)
        return loss

    def configure_optimizers(self):
        optimize edici = torch.optim.Adam(self.parameters(),
        lr=0.001) return optimize edici
```

Şimdi farklı yöntemleri tek tek ele alalım. Gördüğünüz gibi, init kurucusu bir önceki alt bölümde kullandığımız model kodunun aynısını içeriyor. Yeni olan şey, self.train_acc = Accuracy() gibi doğruluk niteliklerini eklemiş olmamızdır. Bunlar, eğitim sırasında doğruluğu izlememizi sağlayacaktır. Accuracy, Lightning ile otomatik olarak yüklenmesi gereken torchmetrics modülünden içe aktarıldı. Torchmetrics'i içe aktaramazsanız, pip install torchmetrics aracılığıyla yüklemeyi deneyebilirsiniz. Daha fazla bilgi https://torchmetrics.readthedocs.io/en/latest/ pages/quickstart.html adresinde bulunabilir.

İleri yöntemi, modelimizi giriş verileri üzerinde çağırdığımızda logitleri (softmax katmanından önce ağımızın son tam bağlı katmanının çıktıları) döndüren basit bir ileri geçiş uygular. İleri yöntemiyle self(x)'i çağırarak hesaplanan logitler, daha sonra açıklayacağımız eğitim, doğrulama ve test adımları için kullanılır.

training_step, training_epoch_end, validation_step, test_step ve configure_optimizers yöntemleri Lightning tarafından özel olarak tanınan yöntemlerdir. Örneğin, training_step de-eğitim sırasında tek bir ileri geçişi cezalandırır, burada doğruluk ve kaybı da takip ederiz, böylece bunları daha sonra analiz edebiliriz. Doğruluğu self.train_acc.update(preds, y) aracılığıyla hesapladığımızı ancak henüz günlüğe kaydetmediğimizi unutmayın. training_step yöntemi, eğitim sırasında her bir yığın üzerinde yürütülür ve her eğitim döneminin sonunda yürütülen training_epoch_end yöntemi aracılığıyla, eğitim yoluyla biriktirdiğimiz doğruluk değerlerinden eğitim kümesi doğruluğunu hesaplarız.

validation_step ve test_step yöntemleri, training_step yöntemine benzer şekilde, doğrulama ve test değerlendirme sürecinin nasıl hesaplanması gerektiğini tanımlar. training_step'e benzer şekilde, her validation_step ve test_step tek bir yığın alır, bu nedenle doğruluğu torchmetric'in Accuracy'sinden türetilen ilgili doğruluk öznitelikleri aracılığıyla kaydederiz. Bununla birlikte, validation_step'in yalnızca belirli aralıklarla, örneğin her eğitim epokundan sonra çağrıldığını unutmayın. Bu nedenle doğrulama doğruluğunu doğrulama adımı içinde günlüğe kaydediyoruz, oysa eğitim doğruluğunu her eğitim döneminden sonra günlüğe kaydediyoruz, aksi takdirde daha sonra incelediğimiz doğruluk grafiği çok gürültülü görünecektir.

Son olarak, configure_optimizers yöntemi aracılığıyla, eğitim için kullanılan optimize ediciyi belirleriz. Aşağıdaki iki alt bölümde veri kümesini nasıl kurabileceğimizi ve modeli nasıl eğitebileceğimizi tartışacağız.

Lightning için veri yükleyicileri ayarlama

Veri kümesini Lightning için hazırlayabileceğimiz üç ana yol vardır. Şunları yapabiliriz:

- Veri kümesini modelin bir parçası haline getirin
- Veri yükleyicileri her zamanki gibi kurun ve bunları bir Lightning Trainer'ın fit yöntemine besleyin - Trainer bir sonraki alt bölümde tanıtılmaktadır
- LightningDataModule oluşturma

Burada, en düzenli yaklaşım olan bir LightningDataModule kullanacağız. Bu LightningDataModule, aşağıda görebileceğimiz gibi beş ana yöntemden oluşur:

```
from torch.utils.data import DataLoader from
t o r c h . utils.data import random_split from
torchvision.datasets import MNIST from
torchvision import transforms

class MnistDataModule(pl.LightningDataModule):
    def init (self, data_path='./'):
        super(). init () self.data_path
        = data_path
        self.transform = t r a n s f o r m s .Compose([transforms.ToTensor()])

    def prepare_data(self):
        MNIST(root=self.data_path, download=True)

    def setup(self, stage=None):
        # aşama ya 'fit', 'validate', 'test' ya da 'predict' #
        burada ilgili not
        mnist_all = MNIST(
            root=self.data_path,
            train=True,
            transform=self.transform,
            download=False
        )

        self.train, self.val = random_split(
            mnist_all, [55000, 5000], generator=torch.Generator().manual_
tohum(1)
        )
```

 olimpikbarbell@gmail.com

```
        self.test = MNIST(
            root=self.data_path,
            train=False,
            transform=self.transform,
            download=False
        )

    def train_dataloader(self):
        return DataLoader(self.train, batch_size=64, num_workers=4)

    def val_dataloader(self):
        return DataLoader(self.val, batch_size=64, num_workers=4)

    def test_dataloader(self):
        return DataLoader(self.test, batch_size=64, num_workers=4)
```

prepare_data yönteminde, veri kümesinin indirilmesi gibi genel adımları tanımlarız. Setup yönteminde, eğitim, doğrulama ve test için kullanılan veri kümelerini tanımlarız. MNIST'in özel bir doğrulama bölmesine sahip olmadığını unutmayın, bu nedenle 60.000 örneklik eğitim setini eğitim için 55.000 örneğe ve doğrulama için 5.000 örneğe bölmek için random_split işlevini kullanıyoruz.

Veri yükleyici yöntemleri kendi kendini açıklar ve ilgili veri kümelerinin nasıl yükleneceğini tanımlar. Şimdi, veri modülünü başlatabilir ve sonraki alt bölümlerde eğitim, doğrulama ve test için

```
torch.manual_seed(1)
mnist_dm = MnistDataModule()
```

kullanabiliriz:

PyTorch Lightning Trainer sınıfını kullanarak modeli eğitme

Artık Lightning veri modülünün yanı sıra özel olarak adlandırılmış yöntemlerle modeli kurmanın ödüllerini toplayabiliriz. Lightning, zero_grad(), backward() ve optimizer.step() gibi tüm ara adımları bizim için hallederek eğitim modelini süper kullanışlı hale getiren bir Trainer sınıfı uygular. Ayrıca, bonus olarak, kullanılacak bir veya daha fazla GPU'yu (varsa) kolayca belirtmemizi sağlar:

```
mnistclassifier = MultiLayerPerceptron()

if torch.cuda.is_available(): # GPU'larınız varsa
    trainer = pl.Trainer(max_epochs=10, gpus=1)
else:
    trainer = pl.Trainer(max_epochs=10)

trainer.fit(model=mnistclassifier, datamodule=mnist_dm)
```

Yukarıdaki kod aracılığıyla, çok katmanlı algılayıcımızı 10 epok için eğitiyoruz. Eğitim sırasında, eğitim ve doğrulama kayıpları gibi epok ve temel metrikleri takip eden kullanışlı bir ilerleme çubuğu görüyoruz:

```
Dönem 9: %100 939/939 [00:07<00:00, 130.42it/s, kayıp=0.1, v_num=0, train_
loss=0.260, valid_loss=0.166, valid_acc=0.949]
```

Eğitim bittikten sonra, bir sonraki alt bölümde göreceğimiz gibi, kaydettiğimiz metrikleri daha ayrıntılı olarak da inceleyebiliriz.

TensorBoard kullanarak modeli değerlendirme

Bir önceki bölümde Trainer sınıfının rahatlığını deneyimledik. Lightning'in bir başka güzel özelliği de loglama yetenekleridir. Daha önce Light- ning modelimizde birkaç self.log adımı belirttiğimizi hatırlayın. Eğitimden sonra ve hatta eğitim sırasında bunları TensorBoard'da görselleştirebiliriz. (Lightning'in diğer kaydedicileri de desteklediğini unutmayın; daha fazla bilgi için lütfen https://pytorch-lightning.readthedocs.io/en/latest/common/loggers.html adresindeki resmi belgelere bakın).

TensorBoard'u Yükleme

TensorBoard, tercihinize bağlı olarak pip veya conda aracılığıyla kurulabilir. Örneğin TensorBoard'u pip ile kurmak için komut aşağıdaki gibidir:

```
pip install tensorboard
```

Lightning'i conda aracılığıyla yüklemek için komut aşağıdadır:

```
conda install tensorboard -c conda-forge
```

Aşağıdaki alt bölümdeki kod, bu komutlarda tensorboard yerine tensorboard==2.4 yazarak yükleyebileceğiniz TensorBoard 2.4 sürümüne dayanmaktadır.

Varsayılan olarak, Lightning eğitimi lightning_logs adlı bir alt klasörde izler. Eğitim çalışmalarını görselleştirmek için, tarayıcınızda TensorBoard'u açacak olan komut satırı terminalinde aşağıdaki kodu çalıştırabilirsiniz:

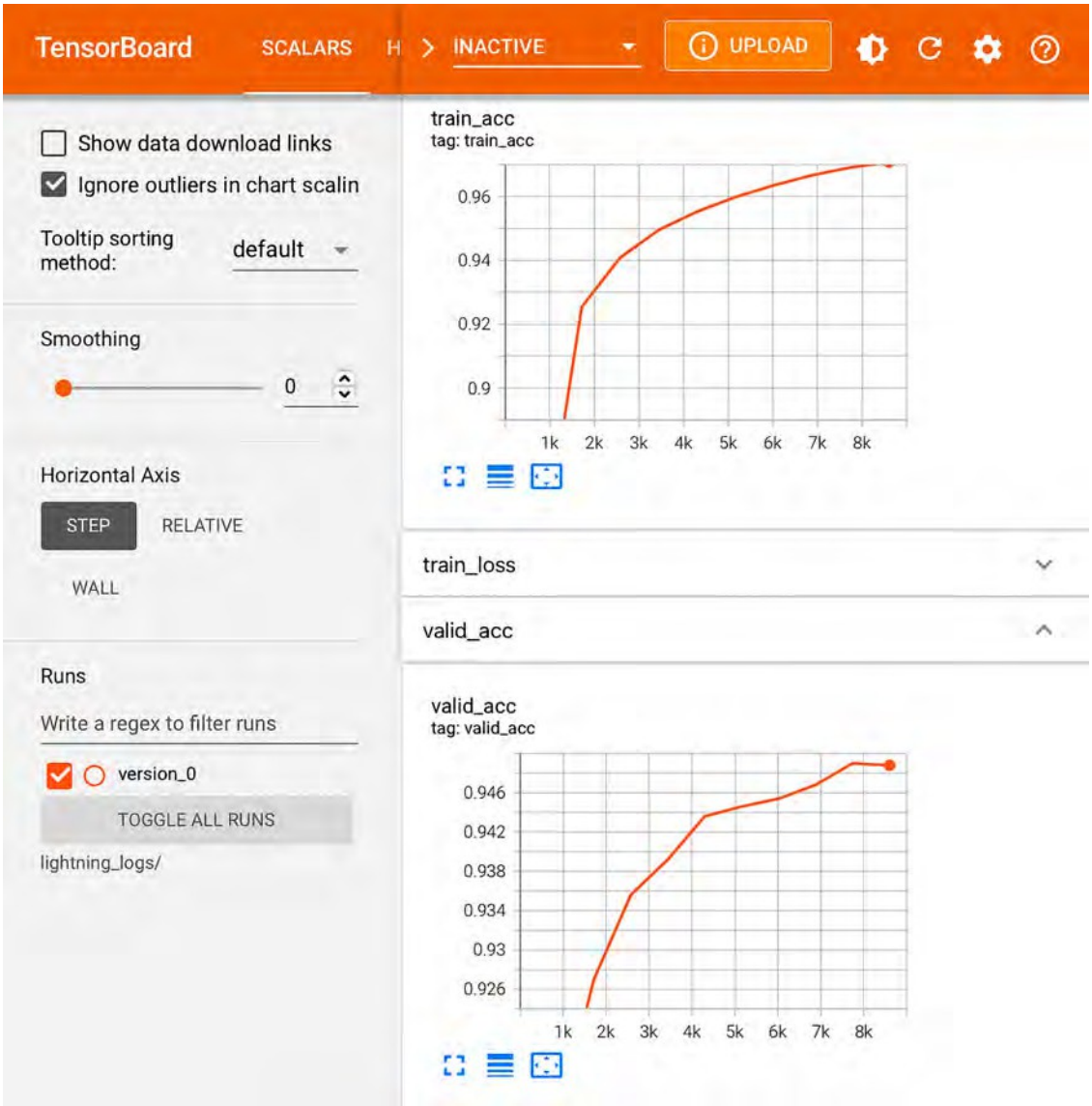
```
tensorboard --logdir lightning_logs/
```

Alternatif olarak, kodu bir Jupyter not defterinde çalıştırıyorsanız, TensorBoard panosunu doğrudan not defterinde göstermek için aşağıdaki kodu bir Jupyter not defteri hücresine ekleyebilirsiniz:

```
load_ext tensorboard
tensorboard --logdir lightning_logs/
```

ilinpikbarbell@gmail.com

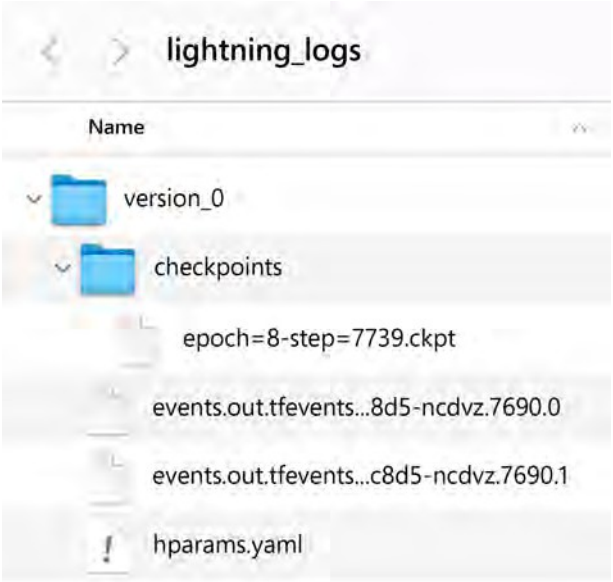
Şekil 13.9, kaydedilen eğitim ve doğrulama doğruluğuyla birlikte TensorBoard kontrol panelini göstermektedir. Sol alt köşede gösterilen bir sürüm_0 geçişi olduğuna dikkat edin. Eğitim kodunu birden fazla kez çalıştırırsanız, Lightning bunları ayrı alt klasörler olarak izleyecektir: version_0, version_1, version_2 ve benzeri:



Şekil 13.9: TensorBoard kontrol paneli

Şekil 13.9'daki eğitim ve doğrulama doğruluklarına bakarak, modeli birkaç ek epok için eğitmenin performansı artırabileceğini varsayabiliriz.

Lightning, eğitilmiş bir modeli yüklememize ve ek epoklar için rahatça eğitmemize olanak tanır. Daha önce de belirtildiği gibi Lightning, alt klasörler aracılığıyla bireysel eğitim çalışmalarını izler. *Şekil 13.10*'da, günlük dosyalarını ve modelin yeniden yüklenmesi için bir model kontrol noktasını içeren version_0 alt klasörünün içeriğini görüyoruz:



Şekil 13.10: PyTorch Lightning günlük dosyaları

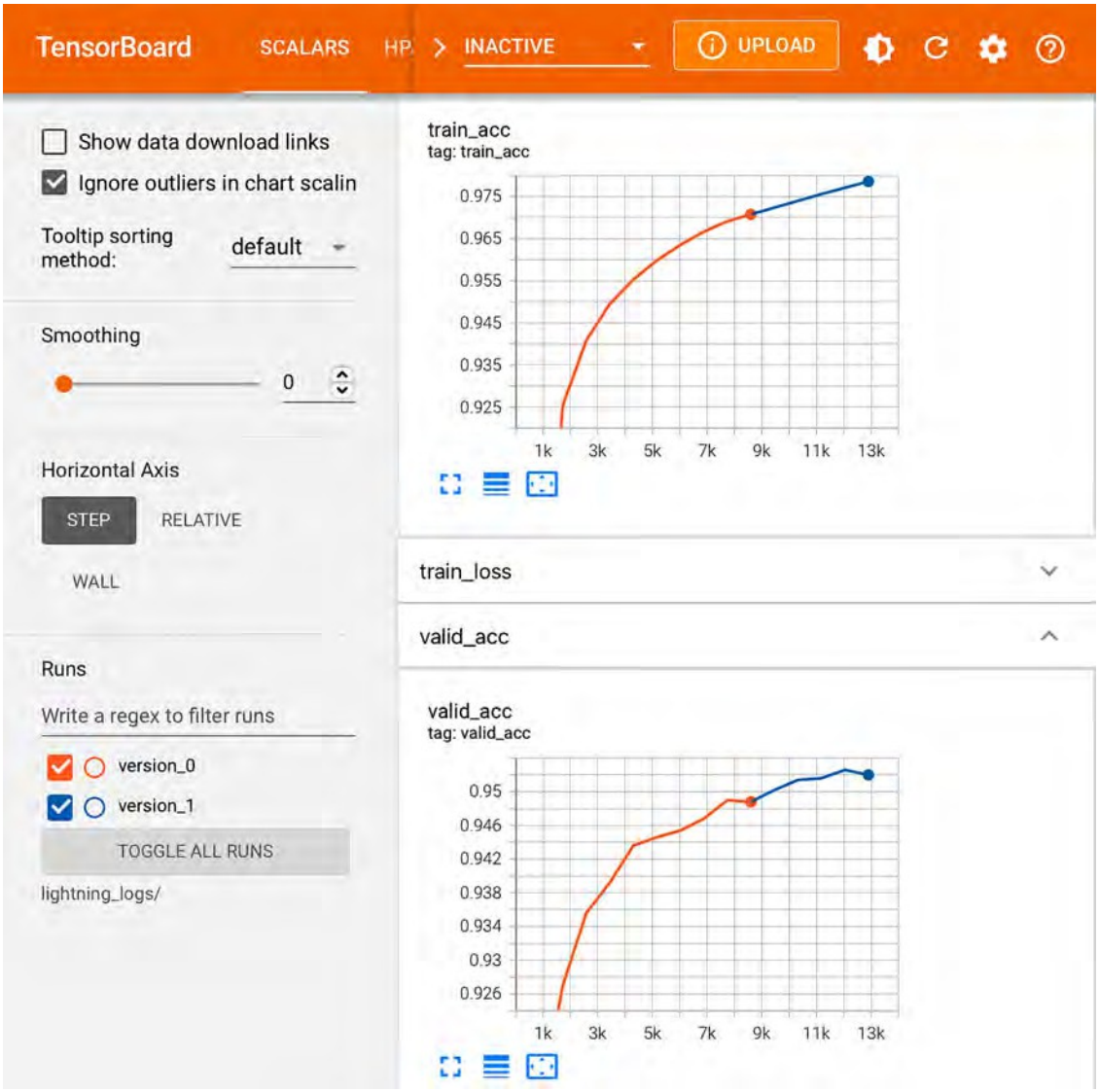
Örneğin, en son model kontrol noktasını bu klasörden yüklemek ve modeli **fit** aracılığıyla eğitmek için aşağıdaki kodu kullanabiliriz:

```
if torch.cuda.is_available(): # GPU'larınız varsa
    trainer = pl.Trainer(max_epochs=15, resume_from_checkpoint='./lightning_
logs/version_0/checkpoints/epoch=8-step=7739.ckpt', gpus=1)
Başka:
    trainer = pl.Trainer(max_epochs=15, resume_from_checkpoint='./lightning_
logs/version_0/checkpoints/epoch=8-step=7739.ckpt')

trainer.fit(model=mnistclassifier, datamodule=mnist_dm)
```

ilinpikbarbell@gmail.com

Burada max_epochs değerini 15 olarak ayarladık ve modeli 5 ek epok için eğittik (daha önce 10 epok için eğitmiştik).

Şimdi *Şekil 13.11*'deki TensorBoard panosuna bir göz atalım ve modeli birkaç ek epok için eğitmenin faydalı olup olmadığını görelim:



Şekil 13.11: TensorBoard gösterge tablosu aker beş epok için daha eğitim

*Şekil 13.11'*de görebileceğimiz gibi, TensorBoard ek eğitim epoklarından (versiyon_1) elde edilen sonuçları öncekilerin (versiyon_0) yanında göstermemize izin verir, ki bu çok kullanışlıdır. Gerçekten de, beş epok daha fazla eğitimin doğrulama doğruluğunu artırdığını görebiliyoruz. Bu noktada, modeli daha fazla epok için eğitmeye karar verebiliriz, bunu size bir alıştırma olarak bırakıyoruz.

Eğitimi tamamladıktan sonra, aşağıdaki kodu kullanarak modeli test seti üzerinde değerlendirebiliriz:

```
trainer.test(model=mnistclassifier, datamodule=mnist_dm)
```

Toplamda 15 epokluk eğitimden sonra elde edilen test seti performansı yaklaşık yüzde 95'tir:

```
[{'test_loss': 0.14912301301956177, 'test_acc': 0.9499600529670715}]
```

PyTorch Lightning'in modeli bizim için otomatik olarak kaydettiğini de unutmayın. Modeli daha sonra tekrar kullanmak isterseniz, aşağıdaki kod aracılığıyla rahatça yükleyebilirsiniz:

```
model = MultiLayerPerceptron.load_from_checkpoint("path/to/checkpoint.ckpt")
```

PyTorch Lightning hakkında daha fazla bilgi edinin

Lightning hakkında daha fazla bilgi edinmek için lütfen https://pytorch-lightning.readthedocs.io adresinde eğitimler ve örnekler içeren resmi web sitesini ziyaret edin.

Lightning ayrıca Slack'te yeni kullanıcıları ve katılımcıları memnuniyetle karşılayan aktif bir topluluğa sahiptir. Daha fazla bilgi edinmek için lütfen https://www.pytorchlightning.ai adresindeki resmi Lightning web sitesini ziyaret edin.

Özet

Bu bölümde PyTorch'un en temel ve kullanışlı özelliklerini ele aldık. PyTorch'un dinamik hesaplama grafiğini tartışarak başladık, bu da hesaplamaların uygulanmasını çok kolay hale getiriyor. Ayrıca PyTorch tensör nesnelerini model parametreleri olarak tanımlama semantiğini de ele aldık.

Keyfi fonksiyonların kısmi türevlerini ve gradyanlarını hesaplama kavramını ele aldıktan sonra, torch.nn modülünü daha ayrıntılı olarak ele aldık. Bu modül bize daha karmaşık derin NN modelleri oluşturmak için kullanıcı dostu bir arayüz sağlamaktadır. Son olarak, şimdiye kadar tartıştıklarımızı kullanarak bir regresyon ve sınıflandırma problemini çözerek bu bölümü tamamladık.

PyTorch'un temel mekaniklerini ele aldığımıza göre, bir sonraki bölümde derin öğrenme için **konvolüsyonel sinir ağı** (**CNN**) mimarilerinin arkasındaki kavramı tanıtacağız. CNN'ler güçlü modellerdir ve bilgisayarla görme alanında büyük performans göstermişlerdir.

olimpikbarbell@gmail.com

Kitabımızın Discord alanına katılın

Benzer düşünen insanlarla tanışmak ve 2000'den fazla üyemizle birlikte öğrenmek için Discord topluluğumuza katılın:
https://packt.link/MLwPyTorch

6587191CA584E
olimpikbarbell@gmail.com

14

Derin Evrişimli Sinir Ağları ile Görüntülerin Sınıflandırılması

Bir önceki bölümde PyTorch sinir ağı ve otomatik farklılaştırma modüllerinin farklı yönlerini derinlemesine inceledik, tensörlere ve dekorasyon fonksiyonlarına aşina oldunuz ve torch.nn ile nasıl çalışacağınızı öğrendiniz. Bu bölümde, şimdi görüntü sınıflandırması için **konvolüsyonel sinir ağları** (**CNN'ler**) hakkında bilgi edineceksiniz. Aşağıdan yukarıya bir yaklaşım kullanarak CNN'lerin temel yapı taşlarını tartışarak başlayacağız. Daha sonra, CNN mimarisine daha derin bir dalış yapacağız ve CNN'lerin PyTorch'ta nasıl uygulanacağını keşfedeceğiz. Bu bölümde aşağıdaki konuları ele alacağız:

- Bir ve iki boyutta konvolüsyon işlemleri
- CNN mimarilerinin yapı taşları
- PyTorch'ta derin CNN'ler uygulama
- Genelleştirme performansını artırmak için veri artırma teknikleri
- Birinin gülümseyip gülümsemediğini tanımak için bir yüz CNN sınıflandırıcısının uygulanması

CNN'lerin yapı taşları

CNN'ler, başlangıçta insan beyninin görsel korteksinin nesneleri tanırken nasıl çalıştığından esinlenen bir model ailesidir. CNN'lerin gelişimi, Yann LeCun ve meslektaşlarının görüntülerden el yazısı rakamları sınıflandırmak için yeni bir NN mimarisi önerdiği 1990'lara kadar uzanmaktadır (*Handwritten Digit Recognition with a Back-Propagation Network* by *Y. LeCun,* and colleagues, 1989, *Neural Information Processing Systems (NeurIPS)* konferansında yayınlanmıştır).

İnsan görsel korteksi

Beynimizin görsel korteksinin nasıl çalıştığına dair orijinal keşif David H. Hubel ve Torsten Wiesel 1959 yılında anestezi uygulanmış bir kedinin birincil görsel korteksine bir mikroelektrot yerleştirdiler. Kedinin önüne farklı ışık desenleri yansıttıktan sonra nöronların farklı tepki verdiğini gözlemlediler. Bu, sonunda görsel korteksin farklı katmanlarının keşfedilmesine yol açtı. Birincil katman esas olarak kenarları ve düz çizgileri algılarken, daha üst düzey katmanlar daha çok karmaşık şekilleri ve desenleri çıkarmaya odaklanır.

CNN'lerin görüntü sınıflandırma görevlerindeki olağanüstü performansı nedeniyle, bu özel ileri beslemeli NN türü büyük ilgi gördü ve bilgisayarla görme için makine öğreniminde muazzam gelişmelere yol açtı. Birkaç yıl sonra, 2019'da Yann LeCun, önceki bölümlerde isimlerine rastladığınız diğer iki araştırmacı Yoshua Bengio ve Geoffrey Hinton ile birlikte **yapay zeka (AI)** alanına yaptığı katkılardan dolayı Turing ödülünü (bilgisayar bilimlerindeki en prestijli ödül) aldı.

Aşağıdaki bölümlerde, CNN'lerin daha geniş kavramlarını ve konvolüsyonel mimarilerin neden genellikle "özellik çıkarma katmanları" olarak tanımlandığını tartışacağız. Daha sonra, CNN'lerde yaygın olarak kullanılan konvolüsyon işleminin teorik tanımına gireceğiz ve bir ve iki boyutta konvolüsyon hesaplama örneklerini inceleyeceğiz.

CNN'leri ve özellik hiyerarşilerini anlama

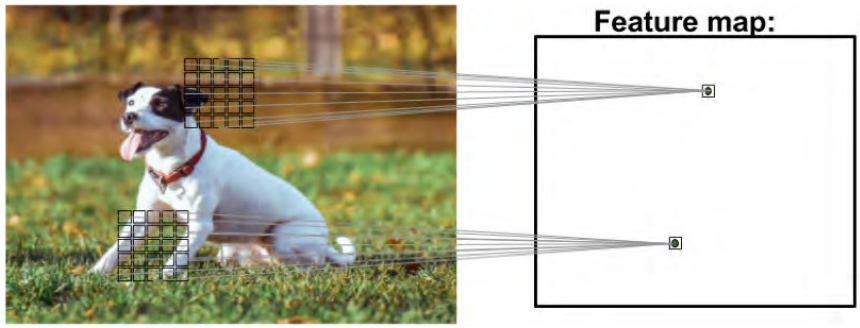
Göze çarpan (ilgili) özelliklerin başarılı bir şekilde çıkarılması, herhangi bir makine öğrenimi algoritmasının performansı için kilit öneme sahiptir ve geleneksel makine öğrenimi modelleri, bir alan uzmanından gelebilecek veya hesaplamalı özellik çıkarma tekniklerine dayanan girdi özelliklerine dayanır.

CNN'ler gibi belirli NN türleri, ham verilerden belirli bir görev için en yararlı olan özellikleri otomatik olarak öğrenebilir. Bu nedenle, CNN katmanlarını özellik çıkarıcılar olarak düşünmek yaygındır: ilk katmanlar (giriş katmanından hemen sonra olanlar) ham verilerden **düşük seviyeli özellikler çıkarır** ve sonraki katmanlar (genellikle **çok katmanlı** bir **algılayıcıda** (**MLP**) olduğu gibi **tam bağlı katmanlar)** bu özellikleri sürekli bir hedef değeri veya sınıf etiketini tahmin etmek için kullanır.

Bazı çok katmanlı NN türleri ve özellikle de derin CNN'ler, yüksek seviyeli özellikler oluşturmak için düşük seviyeli özellikleri katman bazında birleştirerek **özellik hiyerarşisi** adı verilen bir yapı oluşturur. Örneğin, görüntülerle ilgileniyorsak, kenarlar ve lekeler gibi düşük seviyeli özellikler önceki katmanlardan çıkarılır ve bunlar birleştirilerek yüksek seviyeli özellikler oluşturulur. Bu üst düzey özellikler binalar, kediler veya köpekler gibi nesnelerin genel konturları gibi daha karmaşık şekiller oluşturabilir.

alinpikbarbell@gmail.com

Şekil 14.1'de görebileceğiniz gibi, bir CNN, her bir öğenin giriş görüntüsündeki yerel bir piksel yamasından geldiği bir giriş görüntüsünden **özellik haritalarını** hesaplar:



Şekil 14.1: Bir görüntüden özellik haritaları oluşturma (fotoğraf: Alexander Dummer on Unsplash)

Piksellerden oluşan bu yerel yama, **yerel alıcı alan** olarak adlandırılır. CNN'ler genellikle görüntü ile ilgili görevlerde çok iyi performans gösterir ve bu büyük ölçüde iki önemli fikirden kaynaklanır:

- **Seyrek bağlantı**: Özellik haritasındaki tek bir öğe yalnızca küçük bir piksel parçasına bağlıdır. (Bu, MLP'lerde olduğu gibi tüm giriş görüntüsüne bağlanmaktan çok farklıdır. Geriye dönüp *Bölüm 11, Sıfırdan Çok Katmanlı Yapay Sinir Ağı Uygulaması*'nda tüm görüntüye bağlanan tam bağlantılı *bir* ağı nasıl uyguladığımızı karşılaştırmak faydalı olabilir).
- **Parametre paylaşımı**: Giriş görüntüsünün farklı yamaları için aynı ağırlıklar kullanılır.

Bu iki fikrin doğrudan bir sonucu olarak, geleneksel, tam bağlantılı bir MLP'yi bir konvolüsyon katmanıyla değiştirmek, ağdaki ağırlıkların (parametrelerin) sayısını önemli ölçüde azaltır ve göze *çarpan* özellikleri yakalama becerisinde bir iyileşme görürüz. Görüntü verileri bağlamında, yakındaki piksellerin tipik olarak birbirinden uzak piksellere kıyasla birbirleriyle daha alakalı olduğunu varsaymak mantıklıdır.

Tipik olarak, CNN'ler birkaç **konvolüsyonel** ve alt örnekleme katmanından oluşur ve bunları en sonda bir veya daha fazla tam bağlantılı katman takip eder. Tam bağlantılı katmanlar esasen bir MLP'dir; burada her girdi birimi, i, her çıktı birimine, j, ağırlık w ile bağlanır$_{ij}$ (*Bölüm 11'*de daha ayrıntılı olarak ele aldık).

Genellikle **havuzlama katmanları** olarak bilinen alt örnekleme katmanlarının öğrenilebilir parametreleri olmadığını lütfen unutmayın; örneğin, havuzlama katmanlarında ağırlık veya önyargı birimleri yoktur. Ancak, hem evrişimli hem de tam bağlantılı katmanların eğitim sırasında optimize edilen ağırlıkları ve önyargıları vardır.

İlerleyen bölümlerde, konvolüsyon ve havuzlama katmanlarını daha ayrıntılı olarak inceleyecek ve nasıl çalıştıklarını göreceğiz. Konvolüsyon işlemlerinin nasıl çalıştığını anlamak için, bazen metin gibi belirli dizi veri türleriyle çalışmak için kullanılan tek boyutlu bir konvolüsyonla başlayalım. Tek boyutlu konvolüsyonları tartıştıktan sonra, genellikle iki boyutlu görüntülere uygulanan tipik iki boyutlu konvolüsyonlar üzerinde çalışacağız.

Ayrık konvolüsyonlar gerçekleştirme

Ayrık konvolüsyon (ya da sadece **konvolüsyon**) CNN'de temel bir işlemdir. Bu nedenle, bu işlemin nasıl çalıştığını anlamak önemlidir. Bu bölümde, matematiksel tanımı ele alacağız ve bir boyutlu tensörlerin (vektörler) ve iki boyutlu tensörlerin (matrisler) konvolüsyonlarını hesaplamak için bazı **naif** algoritmaları tartışacağız.

Lütfen bu bölümdeki formüllerin ve açıklamaların yalnızca CNN'lerdeki konvolüsyon işlemlerinin nasıl çalıştığını anlamak için olduğunu unutmayın. Aslında, bu bölümün ilerleyen kısımlarında göreceğiniz gibi, PyTorch gibi paketlerde evrişimsel işlemlerin çok daha verimli uygulamaları zaten mevcuttur.

Matematiksel gösterim

Bu bölümde, çok boyutlu bir dizinin boyutunu belirtmek için alt simge kullanacağız (onsor); örneğin, $A_{n_1 \times n_2}$ $n_1 \times n_2$ boyutunda iki boyutlu bir dizidir. Parantezleri, [ ], şunlar için kullanırız: çok boyutlu bir dizinin indekslenmesini ifade eder. Örneğin, $A[i, j]$, A matrisinin i, j indeksindeki elemanını ifade eder. Ayrıca, iki vektör veya matris arasındaki konvolüsyon işlemini belirtmek için özel bir sembol olan $*$ kullandığımızı unutmayın; bu, Python'daki çarpma operatörü * ile karıştırılmamalıdır.

Tek boyutta ayrık konvolüsyonlar

Kullanacağımız bazı temel tanımlar ve notasyonlarla başlayalım. İki vektör, $\boldsymbol{x}$ ve $\boldsymbol{w}$ için ayrık bir konvolüsyon $\boldsymbol{y}$ ile gösterilir; burada $\boldsymbol{x}$ vektörü girişimizdir (bazen **sinyal olarak adlandırılır**) ve $\boldsymbol{w}$ **filtre** veya **çekirdek olarak** adlandırılır. Ayrık bir konvolüsyon matematiksel olarak aşağıdaki gibi tanımlanır:

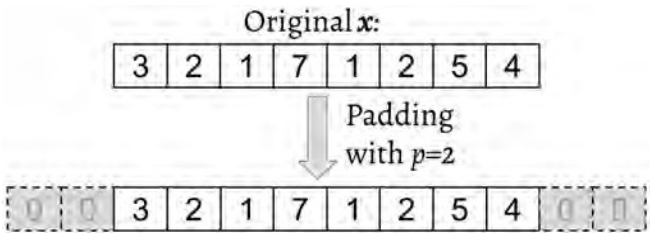
$$\boldsymbol{y}[i] = \sum_{k=-\infty}^{+\infty} \boldsymbol{x}[i-k]\, \boldsymbol{w}[k]$$

Daha önce de belirtildiği gibi, parantezler, [ ], vektör elemanlarının indekslenmesini belirtmek için kullanılır. İndeks, i, çıktı vektörünün her bir elemanı boyunca ilerler, $\boldsymbol{y}$. Önceki formülde açıklığa kavuşturmamız gereken iki garip şey var: $-\infty$ ila $+\infty$ indeksleri ve $\boldsymbol{x}$ için negatif indeksleme.

alinpikbarbell@gmail.com

Toplamın -∞'dan +∞'a kadar olan indislerden geçmesi garip görünmektedir, çünkü makine öğrenimi uygulamalarında her zaman sonlu özellik vektörleriyle uğraşırız. Örneğin, ***x'***in 0, 1, 2, ..., 8, 9 indisli 10 özelliği varsa, -∞: -1 ve 10: +∞ indisleri ***x*** için sınırların dışındadır. Bu nedenle, önceki formülde gösterilen toplamı doğru bir şekilde hesaplamak için, x ve ***w'nin*** sıfırlarla dolu olduğu varsayılır. Bu, aynı zamanda sonsuz boyutta ve çok sayıda sıfır içeren bir çıktı vektörü olan ***y ile*** sonuçlanacaktır. Bu pratik durumlarda kullanışlı olmadığından, ***x*** yalnızca sonlu sayıda sıfırla doldurulur.

Bu işleme **sıfır dolgu** veya basitçe **dolgu** denir. Burada, her bir tarafa eklenen sıfırların sayısı *p* ile gösterilir. ***x*** gibi tek boyutlu bir vektörün örnek bir dolgusu *Şekil 14.2'de* gösterilmiştir:



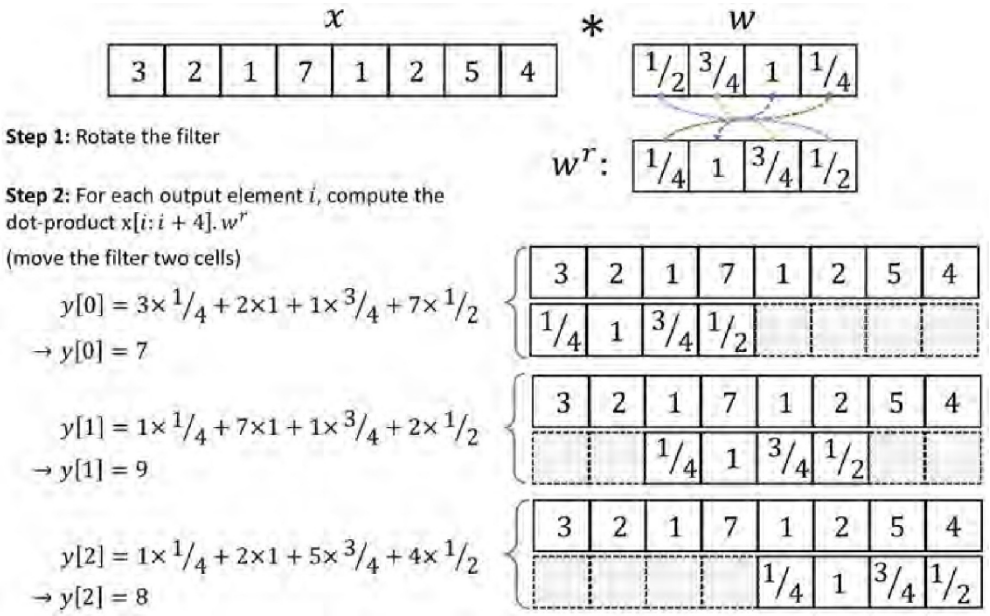
Şekil 14.2: Bir dolgu örneği

Orijinal girişin, ***x***, ve filtrenin, ***w***, sırasıyla *n* ve *m* elemana sahip olduğunu varsayalım, burada *m mn*. Bu nedenle, dolgulu vektör, x^p , *n + 2p* boyutuna sahiptir. Ayrık bir kon- volüsyon hesaplamak için pratik formül aşağıdaki gibi değişecektir:

$$y\,y\,yy\,yy\,y\,y[i]\;y\;\sum_{kkmk}^{kkmk1} x^p\,[i+mi\;k]\,ww[k]$$

Sonsuz indeks sorununu çözdüğümüze göre, ikinci sorun ***x'***$i + m - k$ ile indekslemektir. Burada dikkat edilmesi gereken önemli nokta, bu toplamada ***x*** ve ***w'nin*** farklı yönlerde indekslendiğidir. Toplamı ters yönde giden bir indeksle hesaplamak, bu vektörlerden birini, ***x*** veya ***w,*** doldurulduktan sonra çevirdikten sonra toplamı ileri yönde her iki indeksle hesaplamaya eşdeğerdir. Daha sonra, basitçe bunların nokta çarpımını hesaplayabiliriz. Döndürülmüş filtreyi elde etmek için filtreyi, ***w***, çevirdiğimizi (döndürdüğümüzü) varsayalım, w^r . Daha sonra, nokta çarpımı, $x[i: i + m].w^r$, bir eleman elde etmek için hesaplanır, *y*[i], burada x[*i*: *i* + *m*], ***x'***in *m* boyutlu bir yamasıdır.

Aşağıdaki şekilde $\boldsymbol{x}$ = [3 2 1 7 1 2 5 4] ve $\boldsymbol{www}$ $\left[\frac{1}{2}\ \frac{3}{4}\ 1\ \frac{1}{4}\right]$ ile bir örnek verilmiştir, böylece ilk üç çıkış elemanı hesaplanır:



Şekil 14.3: Ayrık konvolüsyon hesaplama adımları

Önceki örnekte dolgu boyutunun sıfır (p = 0) olduğunu görebilirsiniz. Döndürülmüş filtrenin, $\boldsymbol{w^r}$, her **kaydırdığımızda** iki hücre kaydırıldığına dikkat edin. Bu **kaydırma**, bir konvolüsyonun başka bir hiper parametresidir, adım, s. Bu örnekte adım ikidir, s = 2. Adımın giriş vektörünün boyutundan daha küçük pozitif bir sayı olması gerektiğini unutmayın. Bir sonraki bölümde dolgu ve adımlar hakkında daha fazla konuşacağız.

Çapraz korelasyon

Bir girdi vektörü ile bir filtre arasındaki çapraz korelasyon (veya basitçe korelasyon) y $\boldsymbol{y}$ $\boldsymbol{yy}$ $\boldsymbol{yy}$ ile gösterilir ve küçük bir farkla konvolüsyonun kardeşine çok benzer: çapraz korelasyonda çarpma aynı yönde gerçekleştirilir. Bu nedenle, filtre matrisini, w, her boyutta döndürmek bir gereklilik değildir. Matematiksel olarak çapraz korelasyon aşağıdaki gibi tanımlanır:

$$yy\ \boldsymbol{yy}\ \boldsymbol{y}\ yy\ y\ y[i]\ \boldsymbol{y} \sum_{kkkk\infty}^{+\infty} x[i+k]\ ww[k]$$

Dolgu ve adım için aynı kurallar çapraz korelasyona da uygulanabilir. Çoğu derin öğrenme çerçevesinin (PyTorch dahil) çapraz korelasyonu uyguladığını ancak derin öğrenme alanında yaygın bir gelenek olan konvolüsyon olarak adlandırdığını unutmayın.

alinpikbarbell@gmail.com

Çıktı özellik haritalarının boyutunu kontrol etmek için dolgu girişleri

Şimdiye kadar, sonlu boyutlu çıktı vektörlerini hesaplamak için konvolüsyonlarda yalnızca sıfır dolgu kullandık. Teknik olarak, dolgu herhangi bir *p pp* ile uygulanabilir. *p* seçimine bağlı olarak, sınır hücreleri ***x'in*** ortasında bulunan hücrelerden farklı şekilde ele alınabilir.

Şimdi, $n = 5$ ve $m = 3$ olan bir örnek düşünün. Bu durumda, $p = 0$ olduğunda, $x[0]$ yalnızca bir çıktı elemanının (örneğin *y[0]*) hesaplanmasında kullanılırken, *x[*1] iki çıktı elemanının (örneğin *y[0]* ve *y[1]*) hesaplanmasında kullanılır. Dolayısıyla, x'in elemanlarına yönelik bu farklı muamelenin, çoğu hesaplamada ortaya çıktığı için ortadaki eleman olan $x[2]$'ye yapay olarak daha fazla vurgu yapabileceğini görebilirsiniz. Eğer $p = 2$'yi seçersek bu sorundan kaçınabiliriz, bu durumda ***x'***in her bir elemanı ***y'nin*** üç elemanının hesaplanmasında yer alacaktır.

Ayrıca, çıktının boyutu, ***y***, kullandığımız dolgu stratejisinin seçimine de bağlıdır. Uygulamada

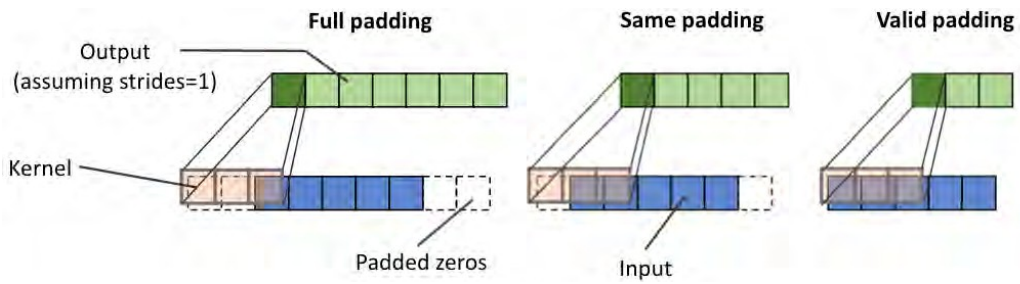
yaygın olarak kullanılan üç dolgu modu vardır: *tam*, *aynı* ve *geçerli*.

Tam modda, dolgu parametresi, *p*, $p = m - 1$ olarak ayarlanır. Tam dolgu, çıktının boyutlarını artırır; bu nedenle CNN mimarilerinde nadiren kullanılır.

Aynı dolgu modu genellikle çıkış vektörünün giriş vektörü ***x*** ile aynı boyutta olmasını sağlamak için kullanılır. Bu durumda, *p* dolgu parametresi, giriş boyutu ve çıkış boyutunun aynı olması gerekliliği ile birlikte filtre boyutuna göre hesaplanır.

Son olarak, geçerli modda bir konvolüsyon hesaplamak $p = 0$ (dolgu yok) durumunu ifade eder.

Şekil 14.4, çekirdek boyutu 3×3 ve adım sayısı 1 olan basit bir 5×5 piksel girişi için üç farklı dolgu modunu göstermektedir:



Şekil 14.4: Üç dolgu modu

CNN'lerde en yaygın kullanılan dolgu modu aynı dolgudur. Diğer dolgu modlarına göre avantajlarından biri, aynı dolgunun vektörün boyutunu veya bilgisayarla görmede görüntü ile ilgili görevler üzerinde çalışırken giriş görüntülerinin yüksekliğini ve genişliğini korumasıdır; bu da bir ağ mimarisi tasarlamayı daha kolay hale getirir.

Tam ve aynı dolguya karşı geçerli dolgunun büyük bir dezavantajı, çok katmanlı NN'lerde tensörlerin hacminin önemli ölçüde azalacak olmasıdır, bu da ağın performansına zarar verebilir. Pratikte, konvolüsyonel katmanlar için aynı dolguyu kullanarak uzamsal boyutu korumalı ve bunun yerine *Striving for Simplicity*'de açıklandığı gibi havuzlama katmanları veya 2 adımlı konvolüsyonel katmanlar yoluyla uzamsal boyutu azaltmalısınız: *The All Convolutional Net* ICLR (workshop track), *Jost Tobias Springenberg, Alexey Dosovitskiy* ve diğerleri, 2015 (https://arxiv.org/abs/1412.6806).

Tam dolguya gelince, boyutu giriş boyutundan daha büyük bir çıkışla sonuçlanır. Tam dolgu genellikle sınır etkilerini en aza indirmenin önemli olduğu sinyal işleme uygulamalarında kullanılır. Ancak, derin öğrenme bağlamında, sınır etkileri genellikle bir sorun değildir, bu nedenle pratikte tam dolgu kullanıldığını nadiren görürüz.

Konvolüsyon çıktısının boyutunu belirleme

Bir konvolüsyonun çıkış boyutu, filtreyi giriş vektörü boyunca kaydırdığımız toplam ***w*** sayısı ile belirlenir. Giriş vektörünün *n* boyutunda ve filtrenin *m* boyutunda olduğunu varsayalım. O zaman, *y* ***y*** *yy y* ***y***'dan kaynaklanan çıktının boyutu, p dolgu *ve s* adımıyla, aşağıdaki gibi belirlenir:

$$o\ o\lfloor\frac{n + np\ nm}{s}\rfloor + 1$$

Burada, $\lfloor\cdot\rfloor$ *zemin* işlemini ifade eder.

Fioor operasyonu

Floor işlemi, örneğin girdiye eşit veya ondan küçük olan en büyük tamsayıyı döndürür:

$$\text{floor}(1.77) = \lfloor 1.77 \rfloor = 1$$

Aşağıdaki iki durumu göz önünde bulundurun:

- Boyutu 5, dolgusu 2 ve adımı 1 olan bir konvolüsyon çekirdeği ile boyutu 10 olan bir girdi vektörü için çıktı boyutunu hesaplayın:

  $$n\ n\ 10, m\ n\ m, p\ n\ m, s\ \ n\ \ 1\ m\ o\ n\ \lfloor\frac{10 + m \times m - m}{1}\rfloor + 1\ \ n\ 10$$

  (Bu durumda, çıktı boyutunun girdiyle aynı olduğuna dikkat edin; bu nedenle, bunun aynı dolgu modu olduğu sonucuna varabiliriz).

- Boyut 3 ve adım 2 olan bir çekirdeğe sahip olduğumuzda aynı girdi vektörü için çıktı boyutu nasıl değişir?

  $$n\ n\ 10, m\ n\ m, p\ n\ m, s\ n\ m\ m\ o\ n\ \lfloor\frac{10 + m \times m - m}{m}\rfloor + 1\ \ n\ 6$$

Konvolüsyon çıktısının boyutu hakkında daha fazla bilgi edinmek istiyorsanız, *Vincent Dumoulin* ve *Francesco Visin* tarafından yazılan ve https://arxiv.org/abs/1603.07285 adresinden ücretsiz olarak erişilebilen *A guide to convolution arithmetic for deep learning adlı* makaleyi tavsiye ederiz.

Son olarak, tek boyutta konvolüsyonların nasıl hesaplanacağını öğrenmek için, aşağıdaki kod bloğunda naif bir uygulama gösterilmekte ve sonuçlar numpy.convolve fonksiyonu ile karşılaştırılmaktadır. Kod aşağıdaki gibidir:

```
>> import numpy as np
>> def conv1d(x, w, p=0, s=1):
...     w_rot = np.array(w[::-1])
...     x_padded = np.array(x)
...     p > 0 ise:
...         zero_pad = np.zeros(shape=p)
...         x_padded = np.concatenate([
...             zero_pad, x_padded, zero_pad
...         ])
...     res = []
...     for i  in range(0, int((len(x_padded) - len(w_rot)) + 1, s):
...         res.append(np.sum(x_padded[i:i+w_rot.shape[0]] * w_rot))
...     return np.array(res)
>> ## Test:
>>> x = [1, 3, 2, 4, 5, 6, 1, 3]
>>> w = [1, 0, 3, 1, 2]
>> print('Conv1d Uygulaması:',
...         conv1d(x, w, p=2, s=1))
Conv1d Uygulaması: [ 5. 14. 16. 26. 24. 34. 19. 22.]
>> print('NumPy Sonuçları:',
...         np.convolve(x, w, mode='same'))
NumPy Sonuçları: [ 5 14 16 26 24 34 19
22]
```

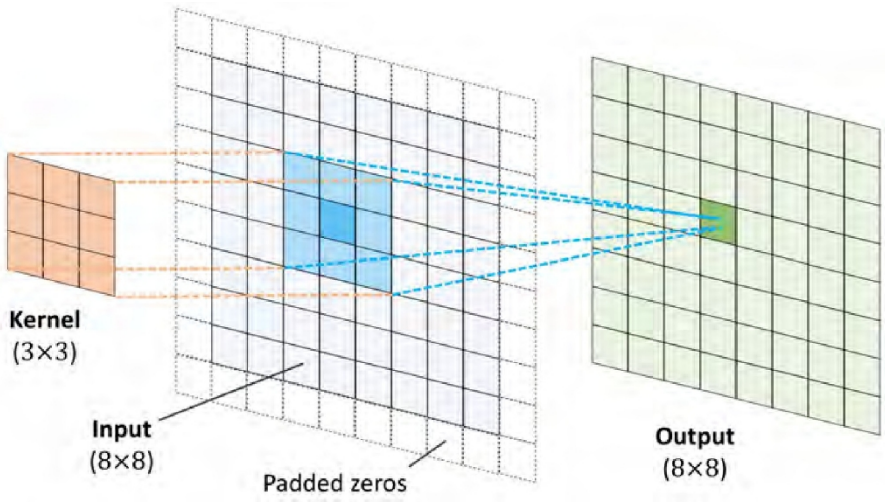
Şimdiye kadar çoğunlukla vektörler için konvolüsyonlara (1D konvolüsyonlar) odaklandık. Kavramların anlaşılmasını kolaylaştırmak için 1B durumla başladık. Bir sonraki bölümde, görüntü ile ilgili görevler için CNN'lerin yapı taşları olan 2D konvolüsyonları daha ayrıntılı olarak ele alacağız.

2B'de ayrık konvolüsyon gerçekleştirme

Önceki bölümlerde öğrendiğiniz kavramlar kolayca 2B'ye genişletilebilir. $XX_{n1\times n2}$ matrisi ve $m \leq n_{11}$ ve $m \leq n_{22}$ olan $WW_{m1\times m2}$ filtre matrisi gibi 2B girdilerle uğraştığımızda, $Y\ Y\ YY\ Y\ YY$ matrisi X ve W arasındaki 2B konvolüsyonun sonucudur. Bu matematiksel olarak aşağıdaki gibi tanımlanır:

$$Y\ Y\ YY\ YY\ Y\ Y[ii\ ii]\ Y \sum_{kk1=-\infty}^{+\infty} \sum_{kk2=-\infty}^{+\infty} X[i\ \ i\ k1ii\ \ i\ k2]\ WW[k1ik2]$$

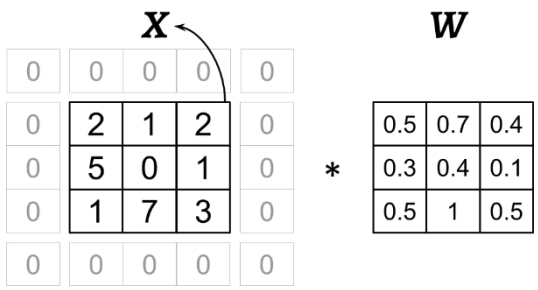
Boyutlardan birini atlarsanız, kalan formülün daha önce 1B'de konvolüsyonu hesaplamak için kullandığımız formülle tamamen aynı olduğuna dikkat edin. Aslında, sıfır dolgu, filtre matrisinin döndürülmesi ve adımların kullanılması gibi daha önce bahsedilen tüm teknikler, her iki boyuta da bağımsız olarak genişletilmeleri koşuluyla, 2B konvolüsyonlar için de geçerlidir. *Şekil 14.5,* 3×3 boyutunda bir çekirdek kullanarak 8×8 boyutunda bir giriş matrisinin 2D konvolüsyonunu göstermektedir. Giriş matrisi $p = 1$ olacak şekilde sıfırlarla doldurulmuştur. Sonuç olarak, 2B konvolüsyonun çıktısı 8×8 boyutunda olacaktır:



Şekil 14.5: 2B konvolüsyonun çıktısı

Aşağıdaki örnekte, $p = (1, 1)$ dolgu ve $s = (2, 2)$ adım kullanılarak bir giriş matrisi olan $\boldsymbol{X}_{3\times3}$ ile bir çekirdek matrisi olan $\boldsymbol{W}_{3\times3}$ arasındaki 2B konvolüsyonun hesaplanması gösterilmektedir. Belirtilen dolguya göre, giriş matrisinin her bir tarafına bir sıfır katmanı eklenir, bu da dolgulu matris ile sonuçlanır

$x_{5\times5}^{padded}$, aşağıdaki gibi:



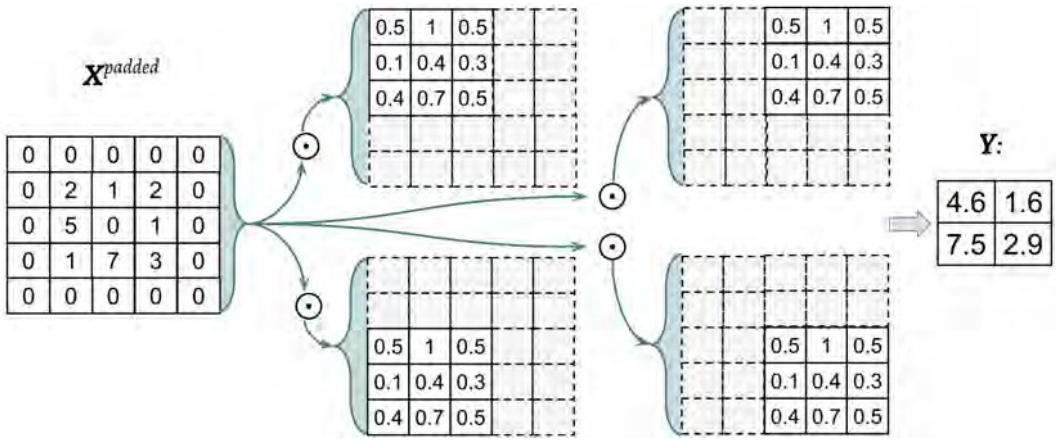
Şekil 14.6: Bir girdi ve çekirdek matrisi arasında 2D konvolüsyon hesaplama

Önceki filtre ile döndürülmüş filtre olacaktır:

$$r\,\boldsymbol{W} = \begin{bmatrix} 0.5 & 1 & 0.5 \\ 0,1 & 0.4 & 0.3 \\ 0.4 & 0.7 & 0.5 \end{bmatrix}$$

alinpikbarbell@gmail.com

Bu rotasyonun transpoze matris ile aynı olmadığını unutmayın. NumPy'de döndürülmüş filtreyi elde etmek için W_rot=W[::-1,::-1] yazabiliriz. Daha sonra, döndürülmüş filtre matrisini dolgulu giriş matrisi X^{padded} boyunca kayan bir pencere gibi kaydırabilir ve *Şekil 14.7*'de $\odot$ operatörü ile gösterilen eleman-bilge çarpımının toplamını hesaplayabiliriz:



Şekil 14.7: Eleman-bilge çarpımının toplamının hesaplanması

Sonuç 2×2 matris, ***Y*** olacaktır.

Ayrıca 2D konvolüsyonu açıklanan *naif* algoritmaya göre uygulayalım. scipy.signal paketi, scipy.signal.convolve2d fonksiyonu aracılığıyla 2D konvolüsyonu hesaplamak için bir yol sağlar:

```
>> import numpy as np
>> import scipy.signal
>> def conv2d(X, W, p=(0, 0), s=(1, 1)):
...     W_rot = np.array(W)[::-1,::-1]
...     X_orig = np.array(X)
...     n1 = X_orig.shape[0] + 2*p[0] n2
...     = X_orig.shape[1] + 2*p[1]
...     X_padded = np.zeros(shape=(n1, n2))
...     X_padded[p[0]:p[0]+X_orig.shape[0],
...                 p[1]:p[1]+X_orig.shape[1]] = X_orig
...
...     res = []
...     for i  in range(0,
...             int((X_padded.shape[0] - \
...             W_rot.shape[0])/s[0])+1, s[0]):
...         res.append([])
```

```
...             for j in range(0,
...                     int((X_padded.shape[1] - \
...                     W_rot.shape[1])/s[1])+1, s[1]):
...                 X_sub = X_padded[i:i+W_rot.shape[0],
...                                 j:j+W_rot.shape[1]]
...                 res[-1].append(np.sum(X_sub * W_rot))
...         return(np.array(res))
>>> X = [[1, 3, 2, 4], [5, 6, 1, 3], [1, 2, 0, 2], [3, 4, 3, 2]]
>>> W = [[1, 0, 3], [1, 2, 1], [0, 1, 1]]
>> print('Conv2d Implementation:\n',
...         conv2d(X, W, p=(1, 1), s=(1, 1))
Conv2d Uygulaması:
[[ 11.   25.   32.   13.]
 [ 19.   25.   24.   13.]
 [ 13.   28.   25.   17.]
 [ 11.   17.   14.    9.]]
>> print('SciPy Sonuçları:\n',
...         scipy.signal.convolve2d(X, W, mode='same'))
SciPy Sonuçları:
[[11 25 32 13]
 [19 25 24 13]
 [13 28 25 17]
 [11 17 14 9]]
```

Konvolüsyon hesaplamak için verimli algoritmalar

Kavramları anlamak amacıyla 2D konvolüsyon hesaplamak için naif bir uygulama sağladık. Ancak, bu uygulama bellek gereksinimleri ve hesaplama karmaşıklığı açısından çok verimsizdir. Bu nedenle, gerçek dünyadaki NN uygulamalarında kullanılmamalıdır.

Bir husus, PyTorch gibi çoğu araçta filtre matrisinin aslında döndürülmemiş olmasıdır. Dahası, son yıllarda konvolüsyonları hesaplamak için Fourier dönüşümünü kullanan çok daha verimli algoritmalar geliştirilmiştir. NN'ler bağlamında, bir konvolüsyon çekirdeğinin boyutunun genellikle giriş görüntüsünün boyutundan çok daha küçük olduğuna dikkat etmek de önemlidir.

Örneğin, modern CNN'ler genellikle 1×1, 3×3 veya 5×5 gibi kernel boyutları kullanır ve bunun için Winograd'ın minimal filtreleme algoritması gibi konvolüsyonel işlemleri çok daha verimli bir şekilde gerçekleştirebilen etkili algoritmalar tasarlanmıştır. Bu algoritmalar bu kitabın kapsamı dışındadır, ancak daha fazla bilgi edinmek isterseniz *Andrew Lavin* ve *Scott Gray*'in 2015 tarihli *Fast Algorithms for Convolutional Neural Networks* adlı makalesini okuyabilirsiniz. https://arxiv.org/abs/1509.09308 adresinden ücretsiz olarak erişilebilir.

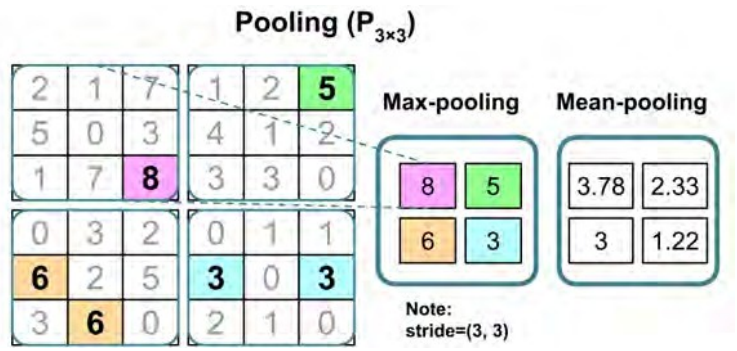
alinpikbarbell@gmail.com

Bir sonraki bölümde, CNN'lerde sıklıkla kullanılan bir diğer önemli işlem olan alt örnekleme veya havuzlama konusunu ele alacağız.

Alt örnekleme katmanları

Alt örnekleme, CNN'lerde tipik olarak iki tür havuzlama işleminde uygulanır: **maksimum havuzlama** ve **ortalama havuzlama** (ortalama havuzlama olarak da bilinir). Havuzlama katmanı genellikle $P_{n1\times n2}$ ile gösterilir. Burada alt simge, maksimum veya ortalama işleminin gerçekleştirildiği komşuluğun boyutunu (her boyuttaki bitişik piksel sayısı) belirler. Böyle bir komşuluğu **havuzlama boyutu** olarak adlandırıyoruz.

İşlem *Şekil 14.8'de* açıklanmıştır. Burada, maksimum havuzlama piksellerin bir komşuluğundan maksimum değeri alır ve ortalama havuzlama bunların ortalamasını hesaplar:



Şekil 14.8: Maksimum havuzlama ve ortalama havuzlama örneği

Havuzlamanın avantajı iki yönlüdür:

- Havuzlama (maksimum havuzlama) yerel bir değişmezlik getirir. Bu, yerel bir komşuluktaki küçük değişikliklerin maksimum havuzlama sonucunu değiştirmediği anlamına gelir. Bu nedenle, giriş verilerindeki gürültüye karşı daha sağlam özellikler üretilmesine yardımcı olur. $\boldsymbol{X}_1$ ve $\boldsymbol{X}_2$ olmak üzere iki farklı giriş matrisinin maksimum havuzlanmasının aynı çıktıyı verdiğini gösteren aşağıdaki örneğe bakın:

$$XX1 = \begin{bmatrix} 10 & 255 & 125 & 0 & 170 & 100 \\ 70 & 255 & 105 & 25 & 25 & 70 \\ 255 & 0 & 150 & 0 & 10 & 10 \\ 0 & 255 & 10 & 10 & 150 & 20 \\ 70 & 15200 & 100 & 950 \\ 35 & 25 & 100 & 20 & 0 & 60 \end{bmatrix}$$

$$\xrightarrow{\text{maksimum havuzlama } P_{2\times2}} \begin{bmatrix} 255 & 125 & 170 \\ 255 & 150 & 150 \\ 70 & 200 & 95 \end{bmatrix}$$

$$XX2 = \begin{bmatrix} 100 & 100 & 100 & 50 & 100 & 50 \\ 95 & 255 & 100 & 125 & 125 & 170 \\ 80 & 40 & 10 & 10 & 125 & 150 \\ 255 & 30 & 150 & 20 & 120 & 125 \\ 30 & 30150 & 100 & 70 & 70 \\ 70 & 30 & 100 & 200 & 70 & 95 \end{bmatrix}$$

- Havuzlama, özelliklerin boyutunu azaltarak daha yüksek hesaplama verimliliği sağlar. Ayrıca, özellik sayısının azaltılması aşırı uyum derecesini de azaltabilir.

Örtüşen ve örtüşmeyen havuzlama

Geleneksel olarak, havuzlamanın örtüşmediği varsayılır. Havuzlama tipik olarak örtüşmeyen komşuluklar üzerinde gerçekleştirilir, bu da stride parametresini ayarlayarak yapılabilir. Örneğin, örtüşmeyen bir havuzlama katmanı, $P_{n1\times n2}$, şunları gerektiren bir adım parametresi $s = (n_1, n_2)$. Öte yandan, adım havuzlama boyutundan daha küçükse örtüşen havuzlama meydana gelir. Çakışan havuzlamanın bir kon- volüsyonel ağda kullanıldığı bir örnek, *A. Krizhevsky*, *I. Sutskever* ve *G. Hinton,* 2012 tarafından hazırlanan ve https://papers.nips.cc/paper/4824-imagenet-classification- with-deep-convolutional-neural-networks adresinden ücretsiz olarak erişilebilen *ImageNet Classification* with Deep Convolutional Neural Networks adlı makalede açıklanmıştır.

Havuzlama hala birçok CNN mimarisinin önemli bir parçası olsa da, havuzlama katmanları kullanılmadan da birkaç CNN mimarisi geliştirilmiştir. Özellik boyutunu azaltmak için havuzlama katmanları kullanmak yerine, araştırmacılar 2 adımlı konvolüsyonel katmanlar kullanmaktadır.

Bir anlamda, adım 2'li bir evrişimsel katmanı, öğrenilebilir ağırlıklara sahip bir havuzlama katmanı olarak düşünebilirsiniz. Havuzlama katmanları ile ve havuzlama katmanları olmadan geliştirilen farklı CNN mimarilerinin ampirik bir karşılaştırması ile ilgileniyorsanız, *Striving for Simplicity adlı* araştırma makalesini okumanızı öneririz: *Jost Tobias Springenberg*, *Alexey Dosovitskiy*, *Thomas Brox* ve *Martin Riedmiller* tarafından yazılan *The All Convolutional Net*. Bu makaleye https://arxiv.org/abs/1412.6806 adresinden ücretsiz olarak erişilebilir.

Her şeyi bir araya getirme - bir CNN uygulama

Şimdiye kadar CNN'lerin temel yapı taşlarını öğrendiniz. Bu bölümde gösterilen kavramlar aslında geleneksel çok katmanlı NN'lerden daha zor değildir. Geleneksel bir NN'deki en önemli işlemin matris çarpımı olduğunu söyleyebiliriz. Örneğin, $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + b'$de olduğu gibi ön aktivasyonları (veya ağ girdilerini) hesaplamak için matris çarpımlarını kullanırız. Burada $\boldsymbol{x}$, pikselleri temsil eden bir sütun vektörü ($\mathbb{R}^{nn1}$ ***matrisi) ve W,*** piksel girdilerini her bir gizli birime bağlayan ağırlık matrisidir.

Bir CNN'de bu işlemin yerini $Z\ Z\ ZZ\ Z\ ZZ+\ b$'de olduğu gibi bir konvolüsyon işlemi alır; burada $\boldsymbol{X}$, *yükseklik*×*genişlik* düzeninde pikselleri temsil eden bir matristir. Her iki durumda da, ön aktivasyonlar bir gizli birimin aktivasyonunu elde etmek için bir aktivasyon fonksiyonuna aktarılır, $A\ A\ \sigma(Z)$, burada σ aktivasyon fonksiyonudur. Ayrıca, alt örneklemenin bir önceki bölümde açıklandığı gibi havuzlama şeklinde ortaya çıkabilen bir CNN'in başka bir yapı taşı olduğunu hatırlayacaksınız.

Birden fazla giriş veya renk kanalı ile çalışma

Bir evrişim katmanının girdisi $N_1 \times N_2$ boyutlarında bir veya daha fazla 2B dizi veya matris içerebilir (örneğin, piksel cinsinden görüntü yüksekliği ve genişliği). Bu $N_1 \times N_2$ matrislere *kanal* adı verilir. Konvolüsyonel katmanların geleneksel uygulamaları girdi olarak bir rank-3 tensör gösterimi bekler, örneğin üç boyutlu bir dizi, $XXN1\times N2\times Cin$, burada C_{in} girdi kanallarının sayısıdır. Örneğin, görüntüleri bir CNN'in ilk katmanına girdi olarak düşünelim. Görüntü renkliyse ve RGB renk modunu kullanıyorsa, $C_{in} = 3$ (RGB'deki kırmızı, yeşil ve mavi renk kanalları için). Ancak, görüntü gri ölçekliyse, $C_{in} = 1$ olur, çünkü gri ölçekli piksel yoğunluk değerlerini içeren yalnızca bir kanal

olimpikbarbell@gmail.com

vardır.

Görüntü dosyası okuma

Görüntülerle çalışırken, örneğin 16 bit, 32 bit veya 64 bit tamsayı türlerine kıyasla bellek kullanımını azaltmak için uint8 (işaretsiz 8 bit tamsayı) veri türünü kullanarak görüntüleri NumPy dizilerine okuyabiliriz.

İşaretsiz 8 bitlik tamsayılar [0, 255] aralığında değerler alır, bu da aynı aralıkta değerler alan RGB görüntülerdeki piksel bilgilerini saklamak için yeterlidir.

Bölüm 12, PyTorch ile Sinir Ağı Eğitimini Paralelleştirme'de PyTorch'un torchvision aracılığıyla görüntüleri yüklemek/saklamak ve işlemek için bir modül sağladığını gördünüz. Bir görüntünün nasıl okunacağını özetleyelim (bu örnek RGB görüntüsü, bu bölümle birlikte verilen kod paketi klasöründe yer almaktadır):

```
>> import torch
>> from torchvision.io import read_image
>> img = read_image('example-image.png')
>> print('Görüntü şekli:', img.shape) Görüntü
şekli: torch.Size([3, 252, 221])
>> print('Kanal sayısı:', img.shape[0]) Kanal sayısı:
3
>> print('Görüntü veri türü:', img.dtype)
Görüntü veri türü: torch.uint8
>> print(img[:, 100:102, 100:102])
  tensör([[179, 182],
          [180, 182]],

         [[134, 136],
          [135, 137]],

         [[110, 112],
          [111, 113]], dtype=torch.uint8)
```

Torchvision ile giriş ve çıkış görüntü tensörlerinin şu formatta olduğunu unutmayın Tensör[channels, image_height, image_width].

Artık girdi verilerinin yapısına aşina olduğunuza göre, bir sonraki soru, önceki bölümlerde tartıştığımız konvolüsyon işlemine birden fazla girdi kanalını nasıl dahil edebileceğimizdir? Cevap çok basittir: her kanal için ayrı ayrı konvolüsyon işlemi gerçekleştirir ve ardından matris toplamını kullanarak sonuçları bir araya getiririz. Her kanalla ilişkili konvolüsyon
(c) $W[:, :, c]$ olarak kendi çekirdek matrisine sahiptir.

alinpikbarbell@gmail.com

Toplam ön aktivasyon sonucu aşağıdaki formülle hesaplanır:

Bir örnek verildiğinde $XX_{n1\times n2\times Cin}$, bir çekirdek matrisi $WW_{m1\times m2\times Cin}$, ve bir yanlılık değeri b $\Rightarrow$

$$Z_{Conv} = \sum_{c=1}^{Cin} WW[:,:,c] * XX[:,:,c]$$

Ön aktivasyon: $Z = Z_{Conv} + b$

Özellik haritası: $A = \sigma(Z)$

Nihai sonuç olan $\boldsymbol{A}$, bir özellik haritasıdır. Genellikle, bir CNN'in evrişimsel katmanında birden fazla özellik haritası bulunur. Birden fazla özellik haritası kullanırsak, çekirdek tensörü dört boyutlu hale gelir: $genişlik \times yükseklik \times C_{in} \times C_{out}$. Burada, $genişlik \times yükseklik$ *çekirdek* boyutu, C_{in} giriş kanallarının sayısı ve C_{out} çıkış özellik haritalarının sayısıdır. Şimdi, çıktı özellik haritalarının sayısını önceki formüle dahil edelim ve aşağıdaki gibi güncelleyelim:

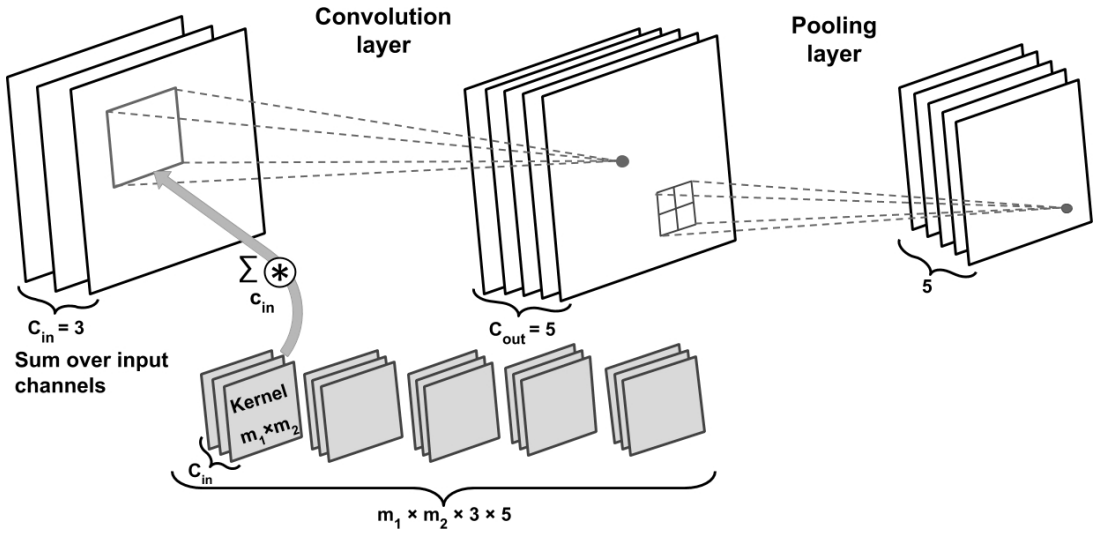
Bir örnek verildiğinde $XX_{n1\times n2\times Cin}$, bir çekirdek matrisi $WW_{m1\times m2\times Cin\times Cout}$, ve bir yanlılık vektörü b_{Cout} $\Rightarrow$

$$Z_{Conv}[:,:,k] = \sum_{c=1}^{Cin} WW[:,:,c,k] * XX[:,:,c]$$

$$Z[:,:,k] = Z_{Conv}[:,:,k] + b[k]$$

$$A[:,:,k] = \sigma(Z[:,:,k])$$

NN'ler bağlamında konvolüsyon hesaplama tartışmamızı sonuçlandırmak için, *Şekil 14.9'*da bir konvolüsyonel katmanı ve ardından bir havuzlama katmanını gösteren örneğe bakalım. Bu örnekte üç giriş kanalı vardır. Çekirdek tensörü dört boyutludur. Her çekirdek matrisi $m_1 \times m_2$ olarak gösterilir ve her giriş kanalı için bir tane olmak üzere bunlardan üç tane vardır. Ayrıca, beş çıktı özellik haritasına karşılık gelen beş çekirdek vardır. Son olarak, özellik haritalarının alt örneklemesi için bir havuzlama katmanı vardır:



Şekil 14.9: Bir CNN'in Uygulanması

Önceki örnekte kaç tane eğitilebilir parametre vardır?

Konvolüsyon, parametre paylaşımı ve seyrek bağlantının avantajlarını göstermek için bir örnek üzerinde çalışalım. *Şekilde* gösterilen ağdaki konvolüsyonel katman *14.9* dört boyutlu bir tensördür. Dolayısıyla, çekirdekle ilişkili $m1 \times m2 \times 3 \times 5$ parametre vardır. Ayrıca, konvolüsyonel katmanın her bir çıktı özellik haritası için bir yanlılık vektörü vardır. Dolayısıyla, yanlılık vektörünün boyutu 5'tir. Havuzlama katmanlarının herhangi bir (eğitilebilir) parametresi yoktur; bu nedenle aşağıdakileri yazabiliriz:

$$_{m1} \times {}_{m2} \times 3 \times 5 + 5$$

Giriş tensörü $_{n1 \times n2 \times 3}$ boyutundaysa, konvolüsyonun aynı dolgu moduyla gerçekleştirildiği varsayılırsa, çıkış özellik haritalarının boyutu $_{n1} \times {}_{n2 \times}$ 5 olacaktır.

Evrişimsel katman yerine tam bağlantılı bir katman kullanırsak bu sayının çok daha büyük olacağını unutmayın. Tam bağlantılı bir katman durumunda, aynı sayıda çıktı birimine ulaşmak için ağırlık matrisinin parametre sayısı aşağıdaki gibi olurdu:

$$(_{n1} \times {}_{n2} \times 3) \times (_{n1} \times {}_{n2 \times} 5) = (_{n1 \times n2})^2 \times 3 \times 5$$

Ayrıca, bias vektörünün boyutu $_{n1} \times {}_{n2} \times$ 5'tir (her çıkış birimi için bir bias elemanı). $_{m1} < {}_{n1}$ ve $_{m2} < {}_{n2}$ olduğu göz önüne alındığında, eğitilebilir parametre sayısındaki farkın önemli olduğunu görebiliriz.

Son olarak, daha önce de belirtildiği gibi, konvolüsyon işlemleri tipik olarak birden fazla renk kanalına sahip bir giriş görüntüsünü bir matris yığını olarak ele alarak gerçekleştirilir; yani, konvolüsyonu her matris üzerinde ayrı ayrı gerçekleştirir ve ardından önceki şekilde gösterildiği gibi sonuçları toplarız. Ancak, örneğin *VoxNet* makalesinde gösterildiği gibi, 3D veri kümeleriyle çalışıyorsanız, konvolüsyonlar 3D hacimlere de genişletilebilir: *Daniel Maturana* ve *Sebastian Scherer*'in 2015 tarihli *A 3D Convolutional Neural Network for Real-Time Object Recognition* başlıklı makalesine https://www.ri.cmu.edu/ pub_files/2015/9/voxnet_maturana_scherer_iros15.pdf adresinden erişilebilir.

Bir sonraki bölümde, bir NN'nin nasıl düzenli hale getirileceğinden bahsedeceğiz.

Bir NN'nin L2 düzenlemesi ve bırakma ile düzenlenmesi

İster geleneksel (tam bağlantılı) bir NN ister bir CNN ile uğraşıyor olalım, bir ağın boyutunu seçmek her zaman zorlu bir sorun olmuştur. Örneğin, makul ölçüde iyi bir performans elde etmek için ağırlık matrisinin boyutunun ve katman sayısının ayarlanması gerekir.

 alinpikbarbell@gmail.com

Bölüm 13, *Going Deeper - The Mechanics of PyTorch*'tan hatırlayacağınız üzere, gizli katmanı olmayan basit bir ağ yalnızca doğrusal bir karar sınırını yakalayabilir, bu da özel veya (veya XOR) veya benzer bir problemle başa çıkmak için yeterli değildir. Bir ağın *kapasitesi,* yaklaşık olarak öğrenebileceği fonksiyonun karmaşıklık seviyesini ifade eder. Küçük ağlar veya nispeten az sayıda parametreye sahip ağlar düşük kapasiteye sahiptir ve bu nedenle karmaşık veri kümelerinin altında yatan yapıyı öğrenemedikleri için *yetersiz uyum* göstererek düşük performansa neden olabilirler. Bununla birlikte, çok büyük ağlar, ağın eğitim verilerini ezberleyeceği ve eğitim veri kümesinde son derece iyi performans gösterirken, tutulan test veri kümesinde zayıf bir performans elde edeceği *aşırı* uyumla sonuçlanabilir. Gerçek dünyadaki makine öğrenimi problemleriyle uğraşırken, ağın ne kadar büyük olması gerektiğini *önceden* bilemeyiz.

Bu sorunu ele almanın bir yolu, eğitim veri kümesinde başarılı olmak için nispeten büyük kapasiteli (pratikte, gerekenden biraz daha büyük bir kapasite seçmek isteriz) bir ağ oluşturmaktır. Ardından, aşırı uyumu önlemek için, test veri kümesi gibi yeni veriler üzerinde iyi bir genelleştirme performansı elde etmek için bir veya birden fazla düzenlilik şeması uygulayabiliriz.

Bölüm 3 ve *4*'te L1 ve L2 düzenlileştirmeyi ele aldık. Her iki teknik de eğitim sırasında ağırlık parametrelerinin küçülmesine neden olan kayba bir ceza ekleyerek aşırı uyumun etkisini önleyebilir veya azaltabilir. Hem L1 hem de L2 düzenlileştirme NN'ler için de kullanılabilirken, L2 bu ikisi arasında daha yaygın bir seçimdir, NN'leri düzenlileştirmek için bu bölümde tartışacağımız bırakma gibi başka yöntemler de vardır. Ancak bırakmayı tartışmaya geçmeden önce, evrişimli veya tam bağlı bir ağda L2 düzenlemesi kullanmak için (hatırlayın, tam bağlı katmanlar PyTorch'ta torch.nn.Linear aracılığıyla uygulanır), PyTorch'taki kayıp fonksiyonuna belirli bir katmanın L2 cezasını aşağıdaki gibi ekleyebilirsiniz:

```
>> import torch.nn as nn
>> kayıp_func = nn.BCELoss()
>> kayıp = kayıp_func(torch.tensor([0.9]), torch.tensor([1.0]))
>> l2_lambda = 0.001
>> conv_layer = nn.Conv2d(in_channels=3,
...                        out_channels=5,
...                        kernel_size=5)
>> l2_ceza = l2_lambda * sum(
...     [(p**2).sum() for p in conv_layer.parameters()]
... )
>> kayıp_ceza ile = kayıp + l2_ceza
>> linear_layer = nn.Linear(10, 16)
```

```
>> l2_ceza = l2_lambda * sum(
...         [(p**2).sum() for p in linear_layer.parameters()]
... )
>> kayıp_ceza ile = kayıp + l2_ceza
```

L2 düzenlemeye karşı ağırlık azalması

L2 düzenlileştirmeyi kullanmanın alternatif bir yolu, örneğin PyTorch optimizerindeki weight_decay parametresini pozitif bir değere ayarlamaktır:

```
optimizer = torch.optim.SGD(
      model.parameters(),
      weight_decay=l2_lambda,
      - - -
)
```

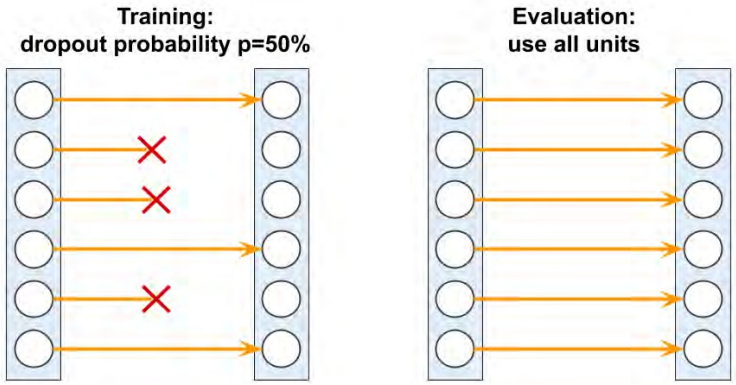
L2 düzenlemesi ve weight_decay tam olarak aynı olmamakla birlikte, **stokastik gradyan inişi** (**SGD**) optimize edicileri kullanıldığında eşdeğer oldukları gösterilebilir. İlgilenen okuyucular daha fazla bilgiyi *Ilya Loshchilov* ve *Frank Hutter'i*n 2019 tarihli *Decoupled Weight Decay Regularization* makalesinde bulabilirler. https://arxiv.org/ adresinden ücretsiz olarak erişilebilir.
abs/1711.05101.

Son yıllarda, **dropout,** aşırı uyumdan kaçınmak için (derin) NN'leri düzenlemek ve böylece genelleme performansını artırmak için popüler bir teknik olarak ortaya çıkmıştır (*Dropout: A Simple Way to Prevent Neural Networks from Overfitting* by *N. Srivastava*, *G. Hinton*, *A. Krizhevsky*, *I. Sutskever*, and *R. Salakhutdin- ov, Journal of Machine Learning Research 15.1*, pages 1929-1958, 2014, http://www.jmlr.org/papers/ volume15/srivastava14a/srivastava14a.pdf). Bırakma genellikle daha yüksek katmanların gizli birimlerine uygulanır ve şu şekilde çalışır: bir NN'nin eğitim aşamasında, gizli birimlerin bir kısmı her iterasyonda p_{drop} olasılığı ile rastgele bırakılır (veya p_{keep} = 1 - p_{drop} olasılığını korur). Bu bırakma olasılığı kullanıcı tarafından belirlenir ve *Nitish Srivastava* ve diğerleri, 2014 tarafından daha önce bahsedilen makalede tartışıldığı gibi yaygın seçim p = 0,5'tir. Giriş nöronlarının belirli bir kısmı bırakıldığında, kalan nöronlarla ilişkili ağırlıklar eksik (bırakılan) nöronları hesaba katmak için yeniden ölçeklendirilir.

Bu rastgele bırakmanın etkisi, ağın verilerin gereksiz bir temsilini öğrenmeye zorlanmasıdır. Bu nedenle ağ, eğitim sırasında herhangi bir zamanda kapatılabilecekleri için herhangi bir gizli birim kümesinin aktivasyonuna güvenemez ve verilerden daha genel ve sağlam modeller öğrenmeye zorlanır.

 alinpikbarbell@gmail.com

Bu rastgele bırakma, aşırı uyumu etkili bir şekilde önleyebilir. *Şekil 14.10,* eğitim aşamasında p = 0,5 olasılıkla bırakma uygulamasının bir örneğini göstermektedir; bu sayede nöronların yarısı rastgele devre dışı kalacaktır (bırakılan birimler eğitimin her ileri geçişinde rastgele seçilir). Ancak tahmin sırasında tüm nöronlar bir sonraki katmanın ön aktivasyonlarının hesaplanmasına katkıda bulunacaktır:



Şekil 14.10: Eğitim aşaması sırasında bırakma uygulaması

Burada gösterildiği gibi, hatırlanması gereken önemli bir nokta, birimlerin yalnızca eğitim sırasında rastgele düşebileceği, oysa değerlendirme (çıkarım) aşaması için tüm gizli birimlerin aktif olması gerektiğidir (örneğin, $p_{drop} = 0$ veya $p_{keep} = 1$). Genel aktivasyonların eğitim ve ön tahmin sırasında aynı ölçekte olmasını sağlamak için, aktif nöronların aktivasyonları uygun şekilde ölçeklendirilmelidir (örneğin, bırakma olasılığı p = 0,5 olarak ayarlanmışsa aktivasyonu yarıya indirerek).

Bununla birlikte, tahminler yaparken aktivasyonları her zaman ölçeklendirmek uygun olmadığından, PyTorch ve diğer araçlar eğitim sırasında aktivasyonları ölçeklendirir (örneğin, bırakma olasılığı p = 0,5 olarak ayarlanmışsa aktivasyonları iki katına çıkararak). Bu yaklaşım genellikle *ters bırakma* olarak adlandırılır.

İlişki hemen açık olmasa da, bırakma, bir model topluluğunun fikir birliği (ortalama) olarak yorumlanabilir. *Bölüm 7*, Topluluk *Öğrenimi için Farklı Modellerin Birleştirilmesi'nde* tartışıldığı gibi, topluluk öğreniminde birkaç modeli bağımsız olarak eğitiriz. Tahmin sırasında, eğitilen tüm modellerin fikir birliğini kullanırız. Model topluluklarının tek modellerden daha iyi performans gösterdiğini zaten biliyoruz. Ancak derin öğrenmede hem birkaç modeli eğitmek hem de birden fazla modelin çıktısını toplamak ve ortalamasını almak hesaplama açısından pahalıdır. Burada dropout, birçok modeli aynı anda eğitmek ve test veya tahmin zamanında ortalama tahminlerini hesaplamak için etkili bir yol ile geçici bir çözüm sunar.

Daha önce de belirtildiği gibi, model toplulukları ve bırakma arasındaki ilişki hemen açık değildir. Ancak, bırakma durumunda her mini parti için farklı bir modelimiz olduğunu düşünün (her ileri geçiş sırasında ağırlıkların rastgele sıfıra ayarlanması nedeniyle).

Ardından, mini gruplar üzerinde yineleme yaparak, esasen $M = 2^h$ modelleri üzerinden örnekleme yaparız; burada h gizli birim sayısıdır.

Bununla birlikte, bırakmayı normal topluluktan ayıran kısıtlama ve özellik, bu "farklı modeller" üzerindeki ağırlıkları paylaşmamızdır, bu da bir tür düzenleme olarak görülebilir. Daha sonra, "çıkarım" sırasında (örneğin, test veri kümesindeki etiketleri tahmin ederken), eğitim sırasında örneklediğimiz tüm bu farklı modellerin ortalamasını alabiliriz. Ancak bu çok pahalıdır.

Daha sonra, modellerin ortalaması, yani bir model tarafından döndürülen sınıf-üyelik olasılığının geometrik ortalamasının hesaplanması, i, aşağıdaki gibi hesaplanabilir:

$$p_{Ensemble} = \left[\prod_{j=1}^{M} p_j(i)\right]^{\frac{1}{M}}$$

Şimdi, bırakmanın arkasındaki *püf noktası,* model topluluklarının (burada *M model*) bu geometrik ortalamasının, eğitim sırasında örneklenen son (veya son) modelin tahminlerini 1/(1 - *p)* faktörü ile ölçeklendirerek yaklaştırılabilmesidir; bu, geometrik ortalamayı önceki denklemi kullanarak açıkça hesaplamaktan çok daha ucuzdur. (Aslında, doğrusal modelleri göz önünde bulundurursak bu yaklaşım tam olarak gerçek geometrik ortalamaya eşdeğerdir).

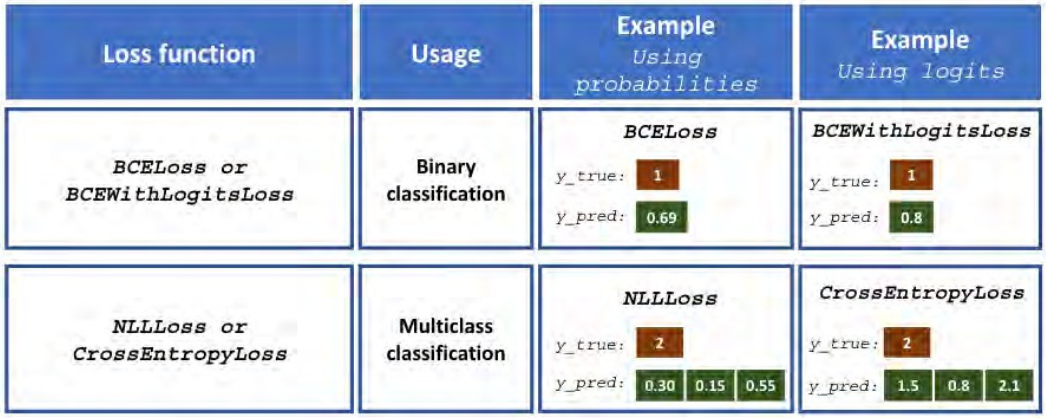
Sınıflandırma için kayıp fonksiyonları

Bölüm 12, *PyTorch ile Sinir Ağı Eğitimini Paralelleştirme'*de ReLU, sigmoid ve tanh gibi farklı aktivasyon fonksiyonlarını gördük. ReLU gibi bu aktivasyon fonksiyonlarından bazıları, modelimize doğrusal olmayan özellikler eklemek için esas olarak bir NN'nin ara (gizli) katmanlarında kullanılır. Ancak sigmoid (ikili için) ve softmax (çoklu sınıf için) gibi diğerleri, modelin çıktısı olarak sınıf-üye-gemi olasılıkları ile sonuçlanan son (çıktı) katmana eklenir. Sigmoid veya softmax aktivasyonları çıkış katmanına dahil edilmezse, model sınıf-üyelik olasılıkları yerine logitleri hesaplayacaktır.

Burada sınıflandırma problemlerine odaklanarak, problemin türüne (ikili ve çok sınıflı) ve çıktı türüne (logitlere karşı olasılıklar) bağlı olarak, modelimizi eğitmek için uygun kayıp fonksiyonunu seçmeliyiz. **İkili çapraz entropi,** ikili sınıflandırma (tek bir çıktı birimi ile) için kayıp fonksiyonudur ve **kategorik çapraz entropi,** çok sınıflı sınıflandırma için kayıp fonksiyonudur. `Torch.nn` modülünde kategorik çapraz entropi kaybı, temel gerçek etiketlerini tam sayı olarak alır (örneğin, *y=2*, üç sınıftan 0, 1 ve 2).

alinpikbarbell@gmail.com

Şekil 14.11'de torch.nn'de her iki durumla başa çıkmak için mevcut olan iki kayıp fonksiyonu açıklanmaktadır: ikili sınıflandırma ve tamsayı etiketli çoklu sınıf. Bu iki kayıp fonksiyonunun her biri, tahminleri logit veya sınıf-üyelik olasılıkları şeklinde alma seçeneğine de sahiptir:



Şekil 14.11: PyTorch'ta iki kayıp fonksiyonu örneği

Çapraz entropi kaybını sınıf-üye-gemi olasılıklarını değil logitleri sağlayarak hesaplamanın sayısal kararlılık nedenleriyle genellikle tercih edildiğini lütfen unutmayın. İkili sınıflandırma için, logitleri nn.BCEWithLogitsLoss() kayıp fonksiyonuna girdi olarak sağlayabilir veya logitlere dayalı olasılıkları hesaplayabilir ve bunları nn.BCELoss() kayıp fonksiyonuna besleyebiliriz. Çok sınıflı sınıflandırma için, logitleri nn.CrossEntropyLoss() kayıp fonksiyonuna girdi olarak sağlayabilir veya logitleri temel alarak log olasılıklarını hesaplayabilir ve bunları nn.NLLLoss() negatif log-likelihood kayıp fonksiyonuna besleyebiliriz.

Aşağıdaki kod size bu kayıp fonksiyonlarını iki farklı formatta nasıl kullanacağınızı gösterecektir; burada logit veya sınıf-üyelik olasılıkları kayıp fonksiyonlarına girdi olarak verilmektedir:

```
>> ####### İkili Çapraz Entropi
>> logits = torch.tensor([0.8])
>>> probas = torch.sigmoid(logits)
>>> hedef = torch.tensor([1.0])
>>> bce_loss_fn = nn.BCELoss()
>>>  bce_logits_loss_fn = nn.BCEWithLogitsLoss()
>> print(f'BCE (w Probas): {bce_loss_fn(probas, target):.4f}') MÖ
(w Probas): 0.3711
>> print(f'BCE (w Logits): '
...          f'{bce_logits_loss_fn(logits, target):.4f}')
BCE (w Logits): 0.3711
```

```
>> ####### Kategorik Çapraz Entropi
>>> logits = torch.tensor([[1.5, 0.8, 2.1]])
>>> probas = torch.softmax(logits, dim=1)
>>> hedef = torch.tensor([2])
>> cce_loss_fn = nn.NLLLoss()
>>  cce_logits_loss_fn = nn.CrossEntropyLoss()
>> print(f'CCE (w Logits): '
...          f'{cce_logits_loss_fn(logits, target):.4f}')
CCE (w Probas): 0.5996
>> print(f'CCE (w Probas): '
...          f'{cce_loss_fn(torch.log(probas), target):.4f}')
CCE (w Logits): 0.5996
```

Bazen ikili sınıflandırma için kategorik çapraz entropi kaybının kullanıldığı bir uygulamaya rastlayabileceğinizi unutmayın. Tipik olarak, ikili bir sınıflandırma görevimiz olduğunda, model her örnek için tek bir çıktı değeri döndürür. Bu tek model çıktısını pozitif sınıfın (örneğin, sınıf 1) olasılığı olarak yorumlarız, $P(\text{sınıf} = \boldsymbol{1}|\boldsymbol{x})$. İkili sınıflandırma probleminde, $P(\text{sınıf} = \boldsymbol{0}|\boldsymbol{x}) = 1 - P(\text{sınıf} = \boldsymbol{1}|\boldsymbol{x})$ olduğu ima edilir; dolayısıyla, negatif sınıfın olasılığını elde etmek için ikinci bir çıktı birimine ihtiyacımız yoktur. Ancak, bazen uygulayıcılar her eğitim örneği için iki çıktı döndürmeyi ve bunları her sınıfın olasılığı olarak yorumlamayı seçerler: $P(\text{sınıf} = \boldsymbol{0}|\boldsymbol{x})$ ve $P(\text{sınıf} = \boldsymbol{1}|\boldsymbol{x})$. Böyle bir durumda, çıktıları normalleştirmek için (toplamları 1 olacak şekilde) bir softmax fonksiyonu (lojistik sigmoid yerine) kullanılması önerilir ve kategorik çapraz entropi uygun kayıp fonksiyonudur.

PyTorch kullanarak derin bir CNN uygulama

Bölüm 13'te, hatırlayacağınız gibi, el yazısı rakam tanıma problemini torch. nn modülünü kullanarak çözmüştük. İki doğrusal gizli katmana sahip bir NN kullanarak yaklaşık yüzde 95,6 doğruluk elde ettiğimizi de hatırlayabilirsiniz.

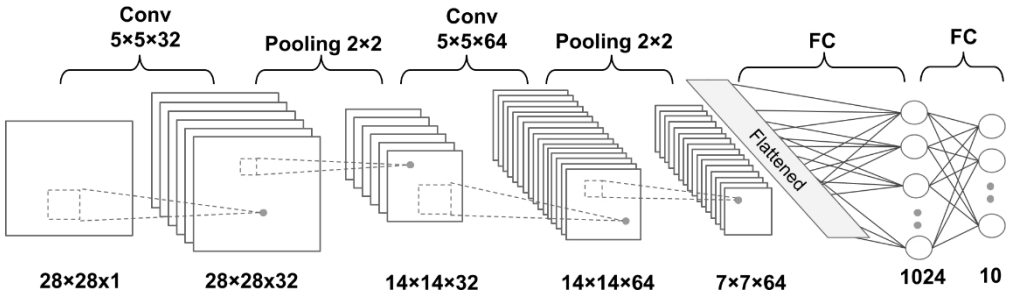
Şimdi, bir CNN uygulayalım ve el yazısı rakamları sınıflandırmak için önceki modele kıyasla daha iyi bir tahmin performansı elde edip edemeyeceğini görelim. *Bölüm 13*'te gördüğümüz tam bağlantılı katmanların bu problemde iyi performans gösterebildiğini unutmayın. Ancak, el yazısı rakamlardan banka hesap numaralarının okunması gibi bazı uygulamalarda, küçük hatalar bile çok maliyetli olabilir. Bu nedenle, bu hatayı mümkün olduğunca azaltmak çok önemlidir.

Çok katmanlı CNN mimarisi

Gerçekleştireceğimiz ağın mimarisi *Şekil 14.12*'de gösterilmiştir. Girdiler 28×28 gri tonlamalı görüntülerdir. Kanal sayısı (gri tonlamalı görüntüler için 1'dir) ve bir yığın giriş görüntüsü göz önüne alındığında, giriş tensörünün boyutları $batchsize \times 28 \times 28 \times 1$ olacaktır.

 alinpikbarbell@gmail.com

Giriş verileri, çekirdek boyutu 5×5 olan iki konvolüsyonel katmandan geçer. İlk konvolüsyon 32 çıktı özellik haritasına, ikincisi ise 64 çıktı özellik haritasına sahiptir. Her bir konvolüsyon katmanını, maksimum havuzlama işlemi şeklinde bir alt örnekleme katmanı takip eder, $P_{2\times2}$. Daha sonra tam bağlantılı bir katman, çıktıyı son *soßmax* çıktı katmanı olarak işlev gören ikinci bir tam bağlantılı katmana aktarır. Uygulayacağımız ağın mimarisi *Şekil 14.12*'de gösterilmektedir:



Şekil 14.12: Derin bir CNN

Her katmandaki tensörlerin boyutları aşağıdaki gibidir:

- Girdi: [*batchsize×28×28×1*]
- Conv_1: [*batchsize×28×28×32*]
- Pooling_1: [batchsize×14×14×32]
- Conv_2: [*batchsize×14×14×64*]
- Pooling_2: [*batchsize×7×7×64*]
- FC_1: [*batchsize×1024*]
- FC_2 ve softmax katmanı: [*batchsize×10*]

Konvolüsyonel çekirdekler için stride=1 kullanıyoruz, böylece giriş boyutları elde edilen özellik haritalarında korunuyor. Havuzlama katmanları için, görüntüyü alt örneklemek ve çıktı özellik haritalarının boyutunu küçültmek için kernel_size=2 kullanıyoruz. Bu ağı PyTorch NN modülünü kullanarak uygulayacağız.

Verilerin yüklenmesi ve ön işlenmesi

İlk olarak, torchvision modülünü kullanarak MNIST veri setini yükleyeceğiz ve *Bölüm 13*'te yaptığımız gibi eğitim ve test setlerini oluşturacağız:

```
>> import torchvision
>> from torchvision import transforms
>> image_path = './'
>> transform = transforms.Compose([
...         transforms.ToTensor()
... ])
```

```
>> mnist_dataset = torchvision.datasets.MNIST(
...         root=image_path, train=True,
...         transform=transform, download=True
... )
>> from torch.utils.data import Subset
>> mnist_valid_dataset = Subset(mnist_dataset,
...                                     torch.arange(10000))
>> mnist_train_dataset = Subset(mnist_dataset,
...                                     torch.arange(
...                                         10000, len(mnist_dataset)
...                                     ))
>> mnist_test_dataset = torchvision.datasets.MNIST(
...         root=image_path, train=False,
...         transform=transform, download=False
... )
```

MNIST veri kümesi önceden belirlenmiş bir eğitim ve test veri kümesi bölümleme şemasıyla birlikte gelir, ancak biz aynı zamanda eğitim bölümünden bir doğrulama bölümü oluşturmak istiyoruz. Bu nedenle, doğrulama için ilk 10.000 eğitim örneğini kullandık. Görüntülerin sınıf etiketine göre sıralanmadığını unutmayın, bu nedenle bu doğrulama seti görüntülerinin aynı sınıflardan olup olmadığı konusunda endişelenmemize gerek yoktur.

Ardından, veri yükleyiciyi sırasıyla eğitim seti ve doğrulama seti için 64 görüntüden oluşan yığınlarla oluşturacağız:

```
>> from torch.utils.data import DataLoader
>>> batch_size = 64
>>> torch.manual_seed(1)
>> train_dl = DataLoader(mnist_train_dataset,
...                             batch_size,
...                             shuffle=True)
>> geçerli_dl = DataLoader(mnist_valid_dataset,
...                             batch_size,
...                             shuffle=False)
```

Okuduğumuz özellikler [0, 1] aralığındaki değerlerden oluşuyor. Ayrıca, görüntüleri zaten tensörlere dönüştürdük. Etiketler 0'dan 9'a kadar tam sayılardır ve on basamağı temsil eder. Dolayısıyla, herhangi bir ölçekleme veya başka bir dönüştürme yapmamıza gerek yoktur.

Şimdi, veri kümesini hazırladıktan sonra, az önce açıkladığımız CNN'i uygulamaya hazırız.

 alinpikbarbell@gmail.com

torch.nn modülünü kullanarak bir CNN uygulama

PyTorch'ta bir CNN uygulamak için, tam bağlantılı katmanların yanı sıra konvolüsyon, havuzlama ve bırakma gibi farklı katmanları istiflemek için torch.nn Sequential sınıfını kullanıyoruz. torch.nn modülü her biri için sınıflar sağlar: iki boyutlu bir konvolüsyon katmanı için nn.Conv2d; alt örnekleme için nn.MaxPool2d ve nn.AvgPool2d (maksimum havuzlama ve ortalama havuzlama); ve dropout kullanarak düzenli hale getirme için nn.Dropout. Bu sınıfların her birinin üzerinden daha ayrıntılı olarak geçeceğiz.

PyTorch'ta CNN katmanlarını yapılandırma

Conv2d sınıfı ile bir katman oluşturmak, çıkış kanallarının sayısını (çıkış özellik haritalarının sayısına veya çıkış filtrelerinin sayısına eşdeğerdir) ve çekirdek boyutlarını belirtmemizi gerektirir.

Ayrıca, bir konvolüsyonel katmanı yapılandırmak için kullanabileceğimiz isteğe bağlı parametreler de vardır. En sık kullanılanlar, her iki boyuttaki örtük dolgu miktarını kontrol eden strides (her iki x, y boyutunda varsayılan değeri 1 olan) ve pad- ding'tir. Ek yapılandırma parametreleri resmi belgelerde listelenmiştir: https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html.

Genellikle bir görüntüyü okuduğumuzda, kanallar için varsayılan boyutun tensör dizisinin ilk boyutu (veya yığın boyutu dikkate alındığında ikinci boyut) olduğunu belirtmek gerekir. Buna *NCHW* formatı denir; burada N yığın içindeki görüntü sayısını, C kanalları, H ve W *ise* sırasıyla yükseklik ve genişliği temsil eder.

Conv2D sınıfının varsayılan olarak girdilerin NCHW biçiminde olduğunu varsaydığını unutmayın. (TensorFlow gibi diğer araçlar NHWC formatını kullanır.) Ancak, kanalları son boyuta yerleştirilmiş bazı verilerle karşılaşırsanız, kanalları birinci boyuta (veya toplu iş boyutunu dikkate alarak ikinci boyuta) taşımak için verilerinizdeki eksenleri değiştirmeniz gerekir. Katman oluşturulduktan sonra, ilk boyutu bir grup örnek için ayrılmış dört boyutlu bir tensör sağlanarak çağrılabilir; ikinci boyut kanala karşılık gelir; ve diğer iki boyut uzamsal boyutlardır.

Oluşturmak istediğimiz CNN modelinin mimarisinde gösterildiği gibi, her bir konvolüsyon katmanını alt örnekleme (özellik haritalarının boyutunu küçültme) için bir havuzlama katmanı takip eder. MaxPool2d ve AvgPool2d sınıfları sırasıyla maksimum havuzlama ve ortalama havuzlama katmanlarını oluşturur. Kernel_size argümanı, maksimum veya ortalama işlemlerini hesaplamak için kullanılacak pencerenin (veya komşuluğun) boyutunu belirler. Ayrıca, stride parametresi daha önce tartıştığımız gibi havuzlama katmanını yapılandırmak için kullanılabilir.

Son olarak, Dropout sınıfı, daha önce tartıştığımız gibi, eğitim sırasında girdi birimlerini bırakma olasılığını belirlemek için kullanılan p_{drop} bırakma olasılığını gösteren p argümanıyla, düzenli hale getirme için bırakma katmanını oluşturacaktır. Bu katman çağrılırken, bu çağrının eğitim sırasında mı yoksa çıkarım sırasında mı yapılacağını belirtmek için model.train() ve model.eval() aracılığıyla davranışı kontrol edilebilir. Dropout kullanırken, bu iki mod arasında geçiş yapmak doğru davranmasını sağlamak için çok önemlidir; örneğin, düğümler yalnızca eğitim sırasında rastgele düşürülür, değerlendirme veya çıkarım sırasında değil.

PyTorch'ta bir CNN Oluşturma

Artık bu sınıflar hakkında bilgi sahibi olduğunuza göre, önceki şekilde gösterilen CNN modelini oluşturabiliriz. Aşağıdaki kodda, Sequential sınıfını kullanacağız ve konvolüsyon ve havuzlama katmanlarını ekleyeceğiz:

```
>>> model = nn.Sequential()
>> model.add_module(
...         'conv1',
...         nn.Conv2d(
...             in_channels=1, out_channels=32,
...             kernel_size=5, padding=2
...         )
... )
>> model.add_module('relu1', nn.ReLU())
>> model.add_module('pool1', nn.MaxPool2d(kernel_size=2))
>> model.add_module(
...         'conv2',
...         nn.Conv2d(
...             in_channels=32, out_channels=64,
...             kernel_size=5, padding=2
...         )
... )
>> model.add_module('relu2', nn.ReLU())
>> model.add_module('pool2', nn.MaxPool2d(kernel_size=2))
```

Şimdiye kadar, modele iki konvolüsyon katmanı ekledik. Her bir evrişim katmanı için 5×5 boyutunda bir çekirdek ve padding=2 kullandık. Daha önce de belirtildiği gibi, aynı dolgu modunu kullanmak, özellik haritalarının uzamsal boyutlarını (dikey ve yatay boyutlar) korur, böylece girişler ve çıkışlar aynı yükseklik ve genişliğe sahip olur (ve kanal sayısı yalnızca kullanılan filtre sayısı açısından farklılık gösterebilir). Daha önce de belirtildiği gibi, çıktı özellik haritasının uzamsal boyutu şu şekilde hesaplanır:

$$o \; o \lfloor \frac{n + np \; nm}{s} \rfloor + 1$$

Burada n giriş özellik haritasının uzamsal boyutudur ve p, m ve s sırasıyla dolguyu, çekirdek boyutunu ve adımı gösterir. $o = i$ *değerini elde etmek için* $p = 2$ değerini elde ediyoruz.

Havuzlama boyutu 2×2 ve adım sayısı 2 olan maksimum havuzlama katmanları uzamsal boyutları yarıya indirecektir. (MaxPool2D'de stride parametresi belirtilmemişse, varsayılan olarak havuzlama çekirdeği boyutuna eşit olarak ayarlandığını unutmayın).

 alinpikbarbell@gmail.com

Bu aşamada özellik haritalarının boyutunu manuel olarak hesaplayabilsek de, PyTorch bunu bizim için hesaplamak için uygun bir yöntem sağlar:

```
>> x = torch.ones((4, 1, 28, 28))
>>> model(x).shape
torch.Size([4, 64, 7, 7])
```

Bu örnekte belirtilen girdi şeklini (4, 1, 28, 28) (yığın içinde 4 görüntü, 1 kanal ve görüntü boyutu 28×28) olarak sağlayarak, çıktının 64 kanallı ve 7×7 uzamsal boyutlu özellik haritalarını gösteren (4, 64, 7, 7) şekline sahip olacağını hesapladık. İlk boyut, keyfi olarak 4 kullandığımız yığın boyutuna karşılık gelir.

Eklemek istediğimiz bir sonraki katman, konvolüsyonel ve havuzlama katmanlarımızın üzerine bir sınıflandırıcı uygulamak için tam bağlantılı bir katmandır. Bu katmanın girdisi rank 2, yani [*batch-size* × *input_units*] şeklinde olmalıdır. Bu nedenle, tam bağlantılı katman için bu gereksinimi karşılamak üzere önceki katmanların çıktılarını düzleştirmemiz gerekir:

```
>> model.add_module('flatten', nn.Flatten())
>> x = torch.ones((4, 1, 28, 28))
>>> model(x).shape
torch.Size([4, 3136])
```

Çıktı şeklinin de gösterdiği gibi, tam bağlı katman için girdi boyutları doğru şekilde ayarlanmıştır. Daha sonra, aralarında bir bırakma katmanı bulunan iki tam bağlantılı katman ekleyeceğiz:

```
>> model.add_module('fc1', nn.Linear(3136, 1024))
>> model.add_module('relu3', nn.ReLU())
>> model.add_module('dropout', nn.Dropout(p=0.5))
>> model.add_module('fc2', nn.Linear(1024, 10))
```

'fc2' olarak adlandırılan son tam bağlı katman, MNIST veri kümesindeki 10 sınıf etiketi için 10 çıkış birimine sahiptir. Pratikte, her bir girdi örneğinin sınıf-üyelik olasılıklarını elde etmek için genellikle sofmax aktivasyonunu kullanırız, sınıfların birbirini dışladığını varsayarız, bu nedenle her bir örneğin olasılıkları 1'e toplanır. Bununla birlikte, softmax işlevi PyTorch'un CrossEntropyLoss uygulaması içinde zaten dahili olarak kullanılmaktadır, bu nedenle yukarıdaki çıktı katmanından sonra bir katman olarak açıkça eklemeniz gerekmez. Aşağıdaki kod, model için kayıp fonksiyonunu

```
>> kayıp_fn = nn.CrossEntropyLoss()
>>> optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

ve optimize ediciyi oluşturacaktır:

Adam iyileştirici

Bu uygulamada, CNN modelini eğitmek için torch.optim.Adam sınıfını kullandığımızı unutmayın. Adam optimizer, konveks olmayan optimizasyon ve makine öğrenimi problemlerine uygun sağlam, gradyan tabanlı bir optimizasyon yöntemidir. İki popüler optimizasyon yöntemi Adam'a ilham vermiştir: RMSProp ve AdaGrad.

Adam'ın en önemli avantajı, gradyan momentlerinin çalışan ortalamasından türetilen güncelleme adımı boyutunun seçimidir. Adam optimize edici hakkında daha fazla bilgi için lütfen *Diederik P. Kingma* ve *Jimmy Lei Ba'nın* 2014 tarihli *Adam*: *A Method for Stochastic Optimization* başlıklı makalesine bakınız. Makaleye https://arxiv.org/abs/1412.6980 adresinden ücretsiz olarak erişilebilir.

Şimdi aşağıdaki fonksiyonu tanımlayarak modeli eğitebiliriz:

```
>> def train(model, num_epochs, train_dl, valid_dl):
...     loss_hist_train = [0] * num_epochs
...     accuracy_hist_train = [0] * num_epochs
...     loss_hist_valid = [0] * num_epochs
...     accuracy_hist_valid = [0] * num_epochs
...     for epoch in range(num_epochs):
...         model.train()
...         for x_batch, y_batch in train_dl:
...             pred = model(x_batch)
...             loss = loss_fn(pred, y_batch)
...             loss.backward()
...             optimizer.step()
...             optimizer.zero_grad()
...             loss_hist_train[epoch] += l o s s .item()*y_batch.size(0)
...             is_correct = (
...                 torch.argmax(pred, dim=1) == y_batch
...             ).float()
...             accuracy_hist_train[epoch] += is_correct.sum()
...         loss_hist_train[epoch] /= len(train_dl.dataset)
...         accuracy_hist_train[epoch] /= len(train_dl.dataset)
...
...         model.eval()
```

 alinpikbarbell@gmail.com

```
...         torch.no_grad() ile:
...             for x_batch, y_batch in valid_dl: pred
...                 = model(x_batch)
...                 loss = loss_fn(pred, y_batch)
...                 loss_hist_valid[epoch] += \
...                     loss.item()*y_batch.size(0)
...                 is_correct = (
...                     torch.argmax(pred, dim=1) == y_batch
...                 ).float()
...                 accuracy_hist_valid[epoch] += is_correct.sum()
...         loss_hist_valid[epoch] /= len(valid_dl.dataset)
...         accuracy_hist_valid[epoch] /= len(valid_dl.dataset)
...
...         print(f'Epoch {epoch+1} doğruluğu: '
...               f'{accuracy_hist_train[epoch]:.4f} val_accuracy: '
...               f'{accuracy_hist_valid[epoch]:.4f}')
...     return loss_hist_train, loss_hist_valid, \
...            accuracy_hist_train, accuracy_hist_valid
```

Eğitim modeli.train() ve değerlendirme modeli.eval() için belirlenen ayarların kullanılmasının, bırakma katmanı için modu otomatik olarak ayarlayacağını ve gizli birimleri uygun şekilde yeniden ölçeklendireceğini, böylece bu konuda endişelenmemize gerek kalmayacağını unutmayın. Daha sonra, bu CNN modelini eğiteceğiz ve öğrenme ilerlemesini izlemek için oluşturduğumuz

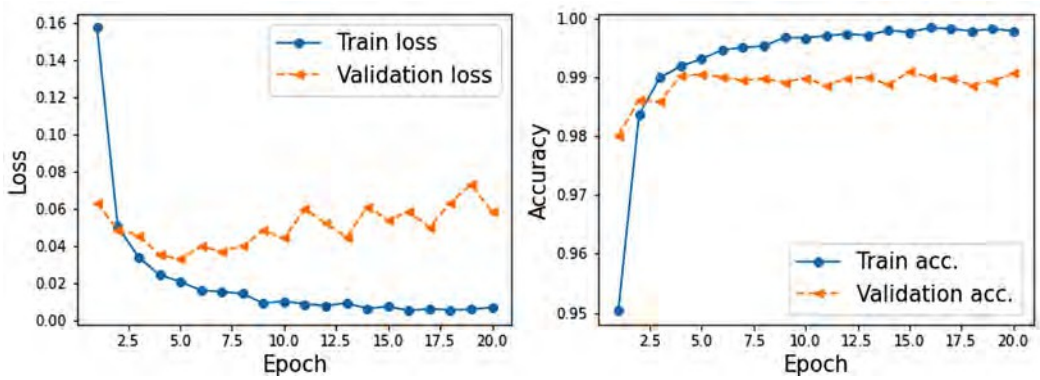
```
>>> torch.manual_seed(1)
>> num_epochs = 20
>> hist = train(model, num_epochs, train_dl, valid_dl)
Dönem 1 doğruluk: 0,9503 val_accuracy: 0,9802
...
Epoch 9 doğruluğu: 0,9968 val_doğruluğu: 0 , 9892
...
Epoch 20 doğruluk: 0,9979 val_accuracy: 0,9907
```

doğrulama veri kümesini kullanacağız:

Eğitimin 20 epoku tamamlandığında, öğrenme eğrilerini görselleştirebiliriz:

```
>> import matplotlib.pyplot as plt
>> x_arr = np.arange(len(hist[0])) + 1
>>> fig = plt.figure(figsize=(12, 4))
>> ax = fig.add_subplot(1, 2, 1)
>> ax.plot(x_arr, hist[0], '-o', label='Tren kaybı')
>> ax.plot(x_arr, hist[1], '--<', label='Doğrulama kaybı')
```

```
>> ax.legend(fontsize=15)
>> ax = fig.add_subplot(1, 2, 2)
>> ax.plot(x_arr, hist[2], '-o', label='Train acc.')
>> ax.plot(x_arr, hist[3], '--<',
...             label='Validation acc.')
>> ax.legend(fontsize=15)
>> ax.set_xlabel('Epoch', size=15)
>>  ax.set_ylabel('Accuracy', size=15)
>> plt.show()
```



Şekil 14.13: Eğitim ve doğrulama verileri için kayıp ve doğruluk grafikleri

Şimdi, eğitilen modeli test veri kümesi üzerinde değerlendiriyoruz:

```
>>> pred = model(mnist_test_dataset.data.unsqueeze(1) / 255.)
>> is_correct = (
...         torch.argmax(pred, dim=1) == mnist_test_dataset.targets
... ).float()
>> print(f'Test doğruluğu: {is_correct.mean():.4f}')
Test doğruluğu: 0,9914
```

CNN modeli yüzde 99,07'lik bir doğruluğa ulaşır. *Bölüm 13*'te yalnızca tam bağlı (evrişimli yerine) katmanlar kullanarak yaklaşık yüzde 95 doğruluk elde ettiğimizi hatırlayın.

Son olarak, tahmin sonuçlarını sınıf-üyelik olasılıkları şeklinde alabilir ve maksimum olasılığa sahip öğeyi bulmak için torch.argmax işlevini kullanarak bunları tahmin edilen etiketlere dönüştürebiliriz. Bunu 12 örnekten oluşan bir grup için yapacağız ve girdi ile tahmin edilen

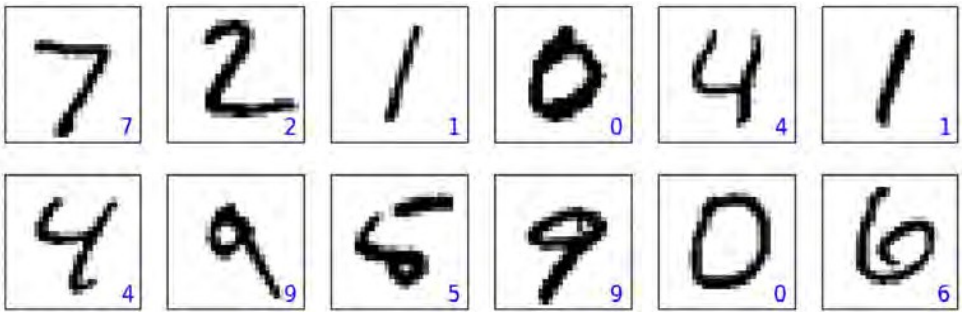
```
>>> fig = plt.figure(figsize=(12, 4))
>> for i  in range(12):
...         ax = fig.add_subplot(2, 6, i+1)
...         ax.set_xticks([]); ax.set_yticks([])
```

etiketleri görselleştireceğiz:

 alinpikbarbell@gmail.com

```
...     img = mnist_test_dataset[i][0][0, :, :]
...     pred = model(img.unsqueeze(0).unsqueeze(1))
...     y_pred = torch.argmax(pred)
...     ax.imshow(img, cmap='gray_r')
...     ax.text(0.9, 0.1, y_pred.item(),
...             size=15, color='blue',
...             horizontalalignment='center',
...             verticalalignment='center',
...             transform=ax.transAxes)
>> plt.show()
```

Şekil 14.14'te el yazısı girdiler ve tahmin edilen etiketleri gösterilmektedir:



Şekil 14.14: El yazısı rakamlar için tahmin edilen etiketler

Çizilen bu örnek kümesinde, tahmin edilen tüm etiketler doğrudur.

Bölüm 11, Sıfırdan Çok Katmanlı *Yapay Sinir Ağı Uygulaması*'nda yaptığımız gibi, yanlış sınıflandırılmış rakamlardan bazılarını gösterme görevini okuyucuya bir alıştırma olarak bırakıyoruz.

CNN kullanarak yüz görüntülerinden gülümseme sınıflandırması

Bu bölümde, CelebA veri kümesini kullanarak yüz görüntülerinden gülümseme sınıflandırması için bir CNN uygulayacağız. *Bölüm 12*'de gördüğünüz gibi, CelebA veri kümesi ünlülerin yüzlerinin 202.599 görüntüsünü içermektedir. Buna ek olarak, her görüntü için bir ünlünün gülümseyip gülümsemediği ve yaşı (genç veya yaşlı) dahil olmak üzere 40 ikili yüz özniteliği mevcuttur.

Şimdiye kadar öğrendiklerinize dayanarak, bu bölümün amacı bu yüz görüntülerinden gülümseme özelliğini tahmin etmek için bir CNN modeli oluşturmak ve eğitmektir. Burada, basitlik açısından, eğitim sürecini hızlandırmak için eğitim verilerinin yalnızca küçük bir bölümünü (16.000 eğitim örneği) kullanacağız. Bununla birlikte, genelleme performansını artırmak ve bu kadar küçük bir veri kümesinde aşırı uyumu azaltmak için **veri artırma** adı verilen bir teknik kullanacağız.

CelebA veri kümesini yükleme

İlk olarak, verileri önceki bölümde MNIST veri kümesi için yaptığımıza benzer şekilde yükleyelim. CelebA verileri üç bölüm halinde gelir: bir eğitim veri kümesi, bir doğrulama veri kümesi ve bir test veri kümesi. Ardından, her bölümdeki örnek sayısını sayacağız:

```
>> image_path = './'
>>  celeba_train_dataset = torchvision.datasets.CelebA(
...         image_path, split='train',
...         target_type='attr', download=True
... )
>> celeba_valid_dataset = torchvision.datasets.CelebA(
...         image_path, split='valid',
...         target_type='attr', download=True
... )
>> celeba_test_dataset = torchvision.datasets.CelebA(
...         image_path, split='test',
...         target_type='attr', download=True
... )
>>>
>> print('Tren seti:', len(celeba_train_dataset)) Tren
seti: 162770
>> print('Doğrulama seti:', len(celeba_valid_dataset))
Doğrulama: 19867
>> print('Test seti:', len(celeba_test_dataset))
Test seti:     19962
```

CelebA veri setini indirmek için alternatif yollar

CelebA veri seti nispeten büyüktür (yaklaşık 1,5 GB) ve torchvision indirme bağlantısı kötü şöhretli bir şekilde istikrarsızdır. Önceki kodu çalıştırmada sorun yaşarsanız, dosyaları resmi CelebA web sitesinden manuel olarak indirebilir (https://mmlab.ie.cuhk.edu.hk/projects/CelebA.html) veya indirme bağlantımızı kullanabilirsiniz: https://drive. google.com/file/d/1m8-EBPgi5MRubrm6iQjafK2QMHDBMSfJ/view?usp=sharing. İndirme bağlantımızı kullanırsanız, kodu çalıştırdığınız geçerli dizinde açmanız gereken bir celeba.zip dosyası indirecektir. Ayrıca, celeba klasörünü indirip açtıktan sonra, yukarıdaki kodu download=True yerine download=False ayarıyla yeniden çalıştırmanız gerekir. Bu yaklaşımla ilgili sorunlarla karşılaşmanız durumunda, size ek bilgi sağlayabilmemiz için lütfen yeni bir sorun açmaktan veya https://github.com/ rasbt/machine-learning-book adresinde bir tartışma başlatmaktan çekinmeyin.

Daha sonra, derin NN'lerin performansını artırmaya yönelik bir teknik olarak veri artırımını tartışacağız.

Görüntü dönüştürme ve veri artırma

Veri artırımı, eğitilmiş verilerin yetersiz olduğu durumlarla başa çıkmak için kullanılan geniş bir

olimpikbarbell@gmail.com

teknikler kümesini özetlemektedir.

veri sınırlıdır. Örneğin, belirli veri artırma teknikleri daha fazla veriyi değiştirmemize ve hatta yapay olarak sentezlememize ve böylece aşırı uyumu azaltarak bir makine veya derin öğrenme modelinin performansını artırmamıza olanak tanır. Veri büyütme yalnızca görüntü verileri için olmasa da, bir görüntünün bazı kısımlarını kırpma, çevirme ve kontrast, parlaklık ve doygunluğu değiştirme gibi görüntü verilerine benzersiz şekilde uygulanabilen bir dizi dönüşüm vardır. Şimdi torchvision.transforms modülü aracılığıyla kullanılabilen bu dönüşümlerden bazılarını görelim. Aşağıdaki kod bloğunda, ilk olarak celeba_train_dataset veri kümesinden beş örnek alacağız ve beş farklı dönüşüm türü uygulayacağız:

1) bir görüntüyü sınırlayıcı bir kutuya kırpmak, 2) bir görüntüyü yatay olarak çevirmek, 3) kontrastı ayarlamak,

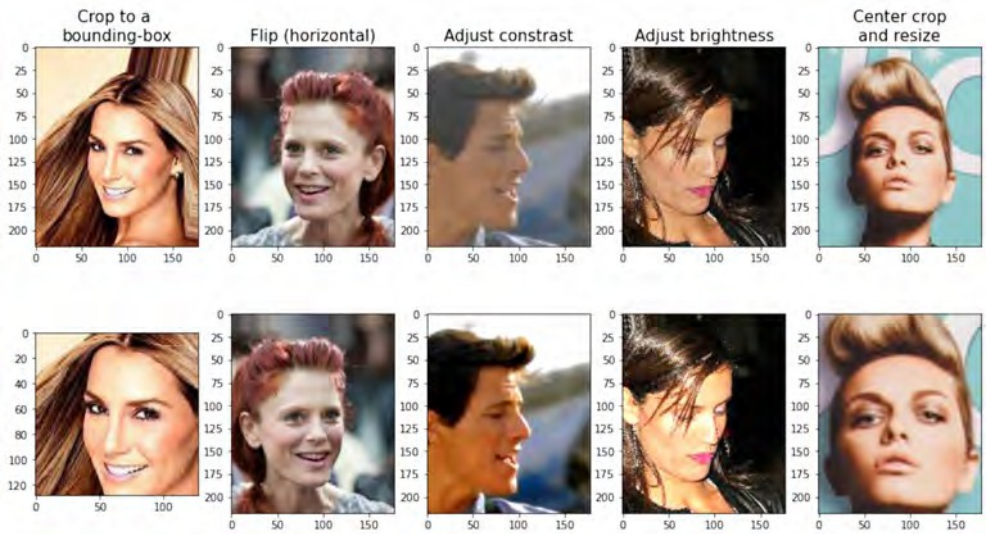
4) parlaklığı ayarlamak ve 5) bir görüntüyü ortalayarak kırpmak ve elde edilen görüntüyü orijinal boyutuna geri boyutlandırmak, (218, 178). Aşağıdaki kodda, karşılaştırma için her birini ayrı bir sütunda göstererek bu dönüşümlerin sonuçlarını görselleştireceğiz:

```
>>> fig = plt.figure(figsize=(16, 8.5))
>> ## Sütun 1: bir sınırlayıcı kutuya kırpma
>> ax = fig.add_subplot(2, 5, 1)
>> img, attr = celeba_train_dataset[0]
>> ax.set_title('Crop to a \nbounding-box', size=15)
>> ax.imshow(img)
>> ax = fig.add_subplot(2, 5, 6)
>> img_cropped = transforms.functional.crop(img, 50, 20, 128, 128)
>> ax.imshow(img_cropped)
>>>
>> ## Sütun 2: çevirme (yatay olarak)
>> ax = fig.add_subplot(2, 5, 2)
>> img, attr = celeba_train_dataset[1]
>> ax.set_title('Flip (yatay)', size=15)
>> ax.imshow(img)
>> ax = fig.add_subplot(2, 5, 7)
> >  img_flipped = transforms.functional.hflip(img)
>> ax.imshow(img_flipped)
>>>
>> ## Sütun 3: kontrastı ayarla
>> ax = fig.add_subplot(2, 5, 3)
>> img, attr = celeba_train_dataset[2]
>> ax.set_title('Adjust constrast', size=15)
>> ax.imshow(img)
>> ax = fig.add_subplot(2, 5, 8)
> >  img_adj_contrast = transforms.functional.adjust_contrast(
...         img, contrast_factor=2
...  )
> >   ax.imshow(img_adj_contrast)
>>>
>> ## Sütun 4: parlaklığı ayarla
>> ax = fig.add_subplot(2, 5, 4)
```

alinpikbarbell@gmail.com

```
>> img, attr = celeba_train_dataset[3]
>> ax.set_title('Parlaklığı ayarla', size=15)
>> ax.imshow(img)
>> ax = fig.add_subplot(2, 5, 9)
> >  img_adj_brightness = transforms.functional.adjust_brightness(
...         img, brightness_factor=1.3
... )
>> ax.imshow(img_adj_brightness)
>>>
>> ## Sütun 5: görüntü merkezinden kırpma
>> ax = fig.add_subplot(2, 5, 5)
>> img, attr = celeba_train_dataset[4]
>> ax.set_title('Center crop\nand resize', size=15)
>> ax.imshow(img)
>> ax = fig.add_subplot(2, 5, 10)
> >  img_center_crop = transforms.functional.center_crop(
...         img, [0.7*218, 0.7*178]
... )
>>  img_resized = transforms.functional.resize(
...         img_center_crop, size=(218, 178)
... )
>> ax.imshow(img_resized)
>> plt.show()
```

Şekil 14.15 sonuçları göstermektedir:



Şekil 14.15: Farklı görüntü dönüşümleri

Şekil 14.15'te orijinal görüntüler ilk satırda, dönüştürülmüş halleri ise ikinci satırda gösterilmektedir.

ikinci satır. İlk dönüşüm için (en soldaki sütun), sınırlayıcı kutunun dört sayı ile belirtildiğine dikkat edin: sınırlayıcı kutunun sol üst köşesinin koordinatı (burada *x=20*, *y=50*) ve kutunun genişliği ve yüksekliği (width=128, height=128). Ayrıca, PyTorch (ve imageio gibi diğer paketler) tarafından yüklenen görüntüler için orijinin ((0, 0) olarak gösterilen konumdaki koordinatlar) görüntünün sol üst köşesi olduğunu unutmayın.

Önceki kod bloğundaki dönüşümler deterministiktir. Ancak, tüm bu dönüşümler rastgele hale de getirilebilir, bu da model eğitimi sırasında veri artırımı için önerilir. Örneğin, bir görüntüden rastgele bir sınırlayıcı kutu (sol üst köşenin koordinatlarının rastgele seçildiği) kırpılabilir, bir görüntü 0,5 olasılıkla yatay veya dikey eksenler boyunca rastgele çevrilebilir veya bir görüntünün kontrastı rastgele değiştirilebilir, burada kontrast_faktörü bir dizi değer arasından rastgele, ancak tek tip dağılımla seçilir. Ek olarak, bu dönüşümlerin bir ardışık düzenini oluşturabiliriz.

Örneğin, bir görüntüyü önce rastgele kırpabilir, sonra rastgele çevirebilir ve son olarak istenen boyuta yeniden boyutlandırabiliriz. Kod aşağıdaki gibidir (rastgele öğelerimiz olduğundan, tekrarlanabilirlik için rastgele tohum ayarlıyoruz):

```
>>> torch.manual_seed(1)
>>> fig = plt.figure(figsize=(14, 12))
>> for i, (img, attr) in enumerate(celeba_train_dataset):
...     ax = fig.add_subplot(3, 4, i*4+1)
...     ax.imshow(img)
...     eğer i == 0 ise:
...         ax.set_title('Orig.', size=15)
...
...     ax = fig.add_subplot(3, 4, i*4+2)
...     img_transform = transforms.Compose([
...         transforms.RandomCrop([178, 178]))
...     ])
...     img_cropped = img_transform(img)
...     ax.imshow(img_cropped)
...     eğer i == 0 ise:
...         ax.set_title('Adım 1: Rastgele kırpma',
...         size=15)
...
...     ax = fig.add_subplot(3, 4, i*4+3) img_transform
...     = transforms.Compose([
...         transforms.RandomHorizontalFlip()
...     ])
...     img_flip = img_transform(img_cropped)
...     ax.imshow(img_flip)
...     eğer i == 0 ise:
...         ax.set_title('Adım 2: Rastgele çevirme',
...         size=15)
```

alinpikbarbell@gmail.com

```
...
...         ax = fig.add_subplot(3, 4, i*4+4)
...         img_resized = transforms.functional.resize(
...             img_flip, size=(128, 128)
...         )
...         ax.imshow(img_resized)
...         eğer i == 0 ise:
...             ax.set_title('Adım 3: Yeniden
...             Boyutlandır', size=15)
...         i == 2
...             ise:
>> plt.show()  Mola
```

Şekil 14.16'da üç örnek görüntü üzerinde rastgele dönüşümler gösterilmektedir:

Şekil 14.16: Rastgele görüntü dönüşümleri

Bu üç örnek üzerinden her yinelemede, rastgele dönüşümler nedeniyle biraz farklı görüntüler elde ettiğimizi unutmayın.

Kolaylık sağlamak için, aşağıdaki işlemler sırasında veri artırımı için bu boru hattını kullanmak üzere dönüştürme işlevleri tanımlayabiliriz

veri kümesi yükleme. Aşağıdaki kodda, 'attributes' listesinden gülümseme etiketini çıkaracak olan get_smile fonksiyonunu tanımlayacağız:

```
>> get_smile = lambda attr: attr[31]
```

Dönüştürülmüş görüntüyü üretecek transform_train fonksiyonunu tanımlayacağız (burada önce görüntüyü rastgele kırpacağız, sonra rastgele çevireceğiz ve son olarak istenen 64×64 boyutuna yeniden boyutlandıracağız):

```
>> transform_train = transforms.Compose([
...         transforms.RandomCrop([178, 178]),
...         transforms.RandomHorizontalFlip(),
...         transforms.Resize([64, 64]),
...         transforms.ToTensor(),
... ])
```

Ancak veri artırımını yalnızca eğitim örneklerine uygulayacağız, doğrulama veya test görüntülerine değil. Doğrulama veya test seti için kod aşağıdaki gibidir (burada önce basitçe görüntüyü kırpacağız ve ardından istenen 64×64 boyutuna yeniden boyutlandıracağız):

```
>> transform = transforms.Compose([
...         transforms.CenterCrop([178, 178]),
...         transforms.Resize([64, 64]),
...         transforms.ToTensor(),
... ])
```

Şimdi, veri artırmayı iş başında görmek için transform_train işlevini eğitim veri kümemize uygulayalım ve veri kümesi üzerinde beş kez yineleyelim:

```
>> from torch.utils.data import DataLoader
>>  celeba_train_dataset = torchvision.datasets.CelebA(
...         image_path, split='train',
...         target_type='attr', download=False,
...         transform=transform_train, target_transform=get_smile
... )
>>> torch.manual_seed(1)
>> data_loader = DataLoader(celeba_train_dataset, batch_size=2)
>>> fig = plt.figure(figsize=(15, 6))
>> num_epochs = 5
>> for j  in range(num_epochs):
...         img_batch, label_batch = next(iter(data_loader))
...         img = img_batch[0]
```

 alinpikbarbell@gmail.com

```
...         ax = fig.add_subplot(2, 5, j + 1)
...         ax.set_xticks([])
...         ax.set_yticks([])
...         ax.set_title(f'Epoch {j}:', size=15)
...         ax.imshow(img.permute(1, 2, 0))
...
...         img = img_batch[1]
...         ax = fig.add_subplot(2, 5, j +
...         6) a x . set_xticks([])
...         ax.set_yticks([])
...         ax.imshow(img.permute(1, 2, 0))
>>> plt.show()
```

Şekil 14.17, iki örnek görüntü üzerinde veri artırımı için ortaya çıkan beş dönüşümü göstermektedir:

Şekil 14.17: Beş görüntü dönüşümünün sonucu

Daha sonra, dönüşüm fonksiyonunu doğrulama ve test veri kümelerimize uygulayacağız:

```
>>> celeba_valid_dataset = torchvision.datasets.CelebA(
...         image_path, split='valid',
...         target_type='attr', download=False,
...         transform=transform, target_transform=get_smile
... )
>>> celeba_test_dataset = torchvision.datasets.CelebA(
...         image_path, split='test',
...         target_type='attr', download=False,
...         transform=transform, target_transform=get_smile
... )
```

Ayrıca, mevcut tüm eğitim ve doğrulama verilerini kullanmak yerine, 16.000 eğitim örneği ve doğrulama için 1.000 örnekten oluşan bir alt küme alacağız, çünkü buradaki amacımız modelimizi kasıtlı olarak küçük bir veri kümesiyle eğitmektir:

```
>> from torch.utils.data import Subset
>> celeba_train_dataset = Subset(celeba_train_dataset,
...                                torch.arange(16000))
>> celeba_valid_dataset = Subset(celeba_valid_dataset,
...                                torch.arange(1000))
>> print('Tren seti:', len(celeba_train_dataset)) Tren
seti: 16000
>> print('Doğrulama seti:', len(celeba_valid_dataset))
Doğrulama seti: 1000
```

Şimdi, üç veri kümesi için veri yükleyicileri oluşturabiliriz:

```
>>> batch_size = 32
>>> torch.manual_seed(1)
>> train_dl = DataLoader(celeba_train_dataset,
...                          batch_size, shuffle=True)
>> geçerli_dl = DataLoader(celeba_valid_dataset,
...                          batch_size, shuffle=False)
>> test_dl = DataLoader(celeba_test_dataset,
...                         batch_size, shuffle=False)
```

Veri yükleyiciler hazır olduğuna göre, bir CNN modeli geliştireceğiz ve bir sonraki bölümde bu modeli eğitip değerlendireceğiz.

Bir CNN gülümseme sınıflandırıcısının eğitimi

Şimdiye kadar, torch.nn modülü ile bir model oluşturmak ve onu eğitmek basit olmalıdır. CNN modelimizin tasarımı şu şekildedir: CNN modeli 3×64×64 boyutunda girdi görüntüleri alır (görüntülerin üç renk kanalı vardır).

Giriş verileri, 3×3 çekirdek boyutuna ve aynı dolgu için 1 dolguya sahip filtreler kullanarak 32, 64, 128 ve 256 özellik haritası oluşturmak için dört konvolüsyonel katmandan geçer. İlk üç konvolüsyon katmanını maksimum havuzlama takip eder, $P_{2\times2}$. Düzenli hale getirme için iki

```
>>> model = nn.Sequential()
>> model.add_module(
...     'conv1',
...     nn.Conv2d(
...         in_channels=3, out_channels=32,
...         kernel_size=3, padding=1
...     )
```

bırakma katmanı da dahil edilmiştir:

 alinpikbarbell@gmail.com

```
... )
>> model.add_module('relu1', nn.ReLU())
>> model.add_module('pool1', nn.MaxPool2d(kernel_size=2))
>> model.add_module('dropout1', nn.Dropout(p=0.5))
>>>
>> model.add_module(
...     'conv2',
...     nn.Conv2d(
...         in_channels=32, out_channels=64,
...         kernel_size=3, padding=1
...     )
... )
>> model.add_module('relu2', nn.ReLU())
>> model.add_module('pool2', nn.MaxPool2d(kernel_size=2))
>> model.add_module('dropout2', nn.Dropout(p=0.5))
>>>
>> model.add_module(
...     'conv3',
...     nn.Conv2d(
...         in_channels=64, out_channels=128,
...         kernel_size=3, padding=1
...     )
... )
>> model.add_module('relu3', nn.ReLU())
>> model.add_module('pool3', nn.MaxPool2d(kernel_size=2))
>>>
>> model.add_module(
...     'conv4',
...     nn.Conv2d(
...         in_channels=128, out_channels=256,
...         kernel_size=3, padding=1
...     )
... )
>> model.add_module('relu4', nn.ReLU())
```

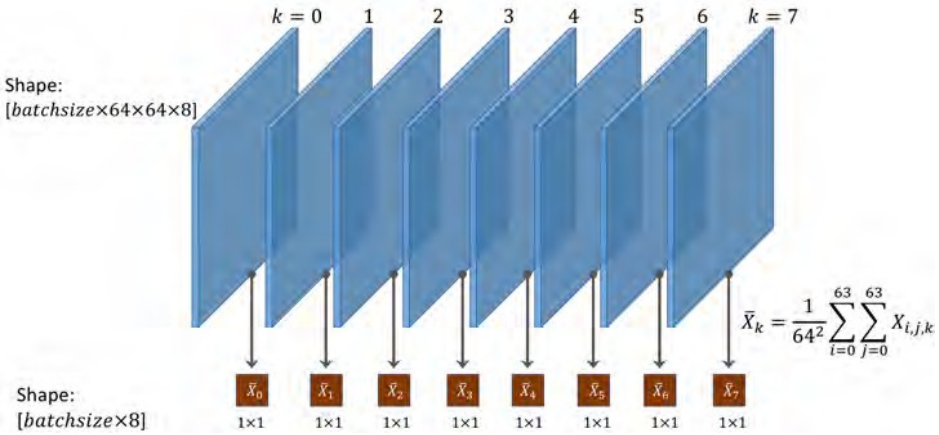
Oyuncak bir toplu girdi (rastgele dört görüntü) kullanarak bu katmanları uyguladıktan sonra çıktı özellik haritalarının şeklini görelim:

```
>> x = torch.ones((4, 3, 64, 64))
>>> model(x).shape
torch.Size([4, 256, 8, 8])
```

8×8 boyutunda 256 özellik haritası (veya kanal) vardır. Şimdi, tek bir birimle çıktı katmanına ulaşmak için tam bağlantılı bir katman ekleyebiliriz. Özellik haritalarını yeniden şekillendirirsek (düzleştirirsek), bu tam bağlantılı katmana giriş birimlerinin sayısı 8 × 8 × 256 = 16.384 olacaktır. Alternatif olarak, her bir özellik haritasının ortalamasını ayrı ayrı hesaplayan ve böylece gizli birimleri 256'ya düşüren *küresel ortalama havuzlama* adı verilen yeni bir katman düşünelim. Daha sonra tam bağlantılı bir katman ekleyebiliriz. Küresel ortalama havuzlama konusunu açıkça ele almamış olsak da, kavramsal olarak diğer havuzlama katmanlarına çok benzemektedir. Aslında küresel ortalama havuzlama, havuzlama boyutu girdi özellik haritalarının boyutuna eşit olduğunda ortalama havuzlamanın özel bir durumu olarak görülebilir.

Bunu anlamak için, *yığın boyutu×8×64×64* şeklindeki girdi özellik haritalarının bir örneğini gösteren *Şekil 14.18'*i düşünün. Kanallar k =0, 1, ..., 7 olarak numaralandırılmıştır. Küresel ortalama-havuzlama işlemi her kanalın ortalamasını hesaplar, böylece çıktı [*batchsize×8*] şekline sahip olur. Bundan sonra, küresel ortalama havuzlama katmanının çıktısını sıkıştıracağız.

Çıktı sıkıştırılmadan, küresel ortalama havuzlama 64×64'lük uzamsal boyutu 1×*1'e* indireceğinden şekil [*batchsize×8×1×1*] olacaktır:



Şekil 14.18: Girdi özellik haritaları

Dolayısıyla, bizim durumumuzda, bu katmandan önceki özellik haritalarının şeklinin [*batchsize×256×8×8*] olduğu göz önüne alındığında, çıktı olarak 256 birim elde etmeyi bekliyoruz, yani çıktının şekli [*batchsize×256*] olacaktır. Bu katmanı ekleyelim ve bunun doğru olduğunu doğrulamak için çıktı

```
>> model.add_module('pool4', nn.AvgPool2d(kernel_size=8))
>> model.add_module('flatten', nn.Flatten())
>> x = torch.ones((4, 3, 64, 64))
>>> model(x).shape
```

şeklini yeniden hesaplayalım:

 alinpikbarbell@gmail.com

```
torch.Size([4, 256])
```

Son olarak, tek bir çıkış birimi elde etmek için tam bağlı bir katman ekleyebiliriz. Bu durumda, aktivasyon fonksiyonunu 'sigmoid' olarak belirleyebiliriz:

```
>> model.add_module('fc', nn.Linear(256, 1))
>> model.add_module('sigmoid', nn.Sigmoid())
>> x = torch.ones((4, 3, 64, 64))
>>> model(x).shape
torch.Size([4, 1])
>> model Sıralı(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (relu1): ReLU()
  (havuz1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
mode=False)
  (dropout1): Dropout(p=0.5, inplace=False)
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (relu2): ReLU()
  (havuz2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
mode=False)
  (dropout2): Dropout(p=0.5, inplace=False)
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (relu3): ReLU()
  (havuz3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
mode=False)
  (conv4): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (relu4): ReLU()
  (havuz4): AvgPool2d(kernel_size=8, stride=8, padding=0)
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (fc): Linear(in_features=256, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

Bir sonraki adım bir kayıp fonksiyonu ve optimizer (Adam optimizer tekrar) oluşturmaktır. Tek bir olasılıksal çıktıya sahip ikili bir sınıflandırma için, kayıp fonksiyonu olarak BCELoss kullanıyoruz:

```
>> kayıp_fn = nn.BCELoss()
>>> optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Şimdi aşağıdaki fonksiyonu tanımlayarak modeli eğitebiliriz:

```
>> def train(model, num_epochs, train_dl, valid_dl):
...     loss_hist_train = [0] * num_epochs
...     accuracy_hist_train = [0] * num_epochs
...     loss_hist_valid = [0] * num_epochs
...     accuracy_hist_valid = [0] * num_epochs
...     for epoch in range(num_epochs):
...         model.train()
...         for x_batch, y_batch in train_dl:
...             pred = model(x_batch)[:, 0]
...             kayıp = kayıp_fn(pred, y_batch.float())
...             loss.backward()
...             optimizer.step()
...             optimizer.zero_grad()
...             loss_hist_train[epoch] += l o s s .item()*y_batch.size(0)
...             is_correct = ((pred>=0.5).float() == y_batch).float()
...             accuracy_hist_train[epoch] += is_correct.sum()
...         loss_hist_train[epoch] /= len(train_dl.dataset)
...         accuracy_hist_train[epoch] /= len(train_dl.dataset)
...
...         model.eval()
...         torch.no_grad() ile:
...             for x_batch, y_batch in valid_dl:
...                 pred = model(x_batch)[:, 0]
...                 kayıp = kayıp_fn(pred, y_batch.float())
...                 loss_hist_valid[epoch] += \
...                     loss.item() * y_batch.size(0)
...                 is_correct = \
...                     ((pred>=0.5).float() == y_batch).float()
...                 accuracy_hist_valid[epoch] += is_correct.sum()
...         loss_hist_valid[epoch] /= len(valid_dl.dataset)
...         accuracy_hist_valid[epoch] /= len(valid_dl.dataset)
...
...         print(f'Epoch {epoch+1} doğruluğu: '
...               f'{accuracy_hist_train[epoch]:.4f} val_accuracy: '
...               f'{accuracy_hist_valid[epoch]:.4f}')
...     return loss_hist_train, loss_hist_valid, \
...            accuracy_hist_train, accuracy_hist_valid
```

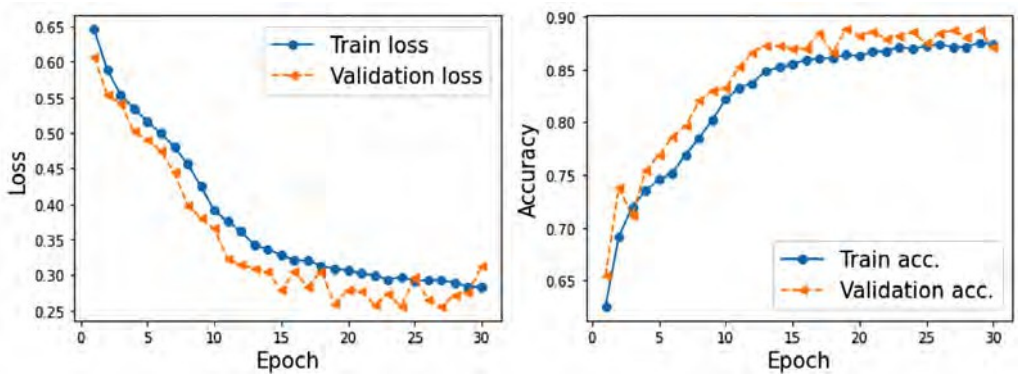
 alinpikbarbell@gmail.com

Daha sonra, bu CNN modelini 30 epok boyunca eğiteceğiz ve öğrenme ilerlemesini izlemek için oluşturduğumuz doğrulama veri kümesini kullanacağız:

```
>>> torch.manual_seed(1)
>> num_epochs = 30
>> hist = train(model, num_epochs, train_dl, valid_dl) Epoch
1 doğruluk: 0,6286 val_accuracy: 0 , 6540
---
Epoch 15 doğruluk: 0,8544 val_accuracy: 0 , 8700
---
Epoch 30 doğruluk: 0,8739 val_accuracy: 0,8710
```

Şimdi öğrenme eğrisini görselleştirelim ve her epoktan sonra eğitim ve doğrulama kayıplarını ve doğruluklarını karşılaştıralım:

```
>> x_arr = np.arange(len(hist[0])) + 1
>>> fig = plt.figure(figsize=(12, 4))
>> ax = fig.add_subplot(1, 2, 1)
>> ax.plot(x_arr, hist[0], '-o', label='Tren kaybı')
>> ax.plot(x_arr, hist[1], '--<', label='Doğrulama kaybı')
>> ax.legend(fontsize=15)
>> ax = fig.add_subplot(1, 2, 2)
>> ax.plot(x_arr, hist[2], '-o', label='Train acc.')
>> ax.plot(x_arr, hist[3], '--<',
...             label='Validation acc.')
>> ax.legend(fontsize=15)
>> ax.set_xlabel('Epoch', size=15)
>>  ax.set_ylabel('Accuracy', size=15)
>> plt.show()
```



Şekil 14.19: Eğitim ve doğrulama sonuçlarının karşılaştırılması

Öğrenme eğrilerinden memnun olduğumuzda, modeli bekletilen test veri kümesi üzerinde değerlendirebiliriz:

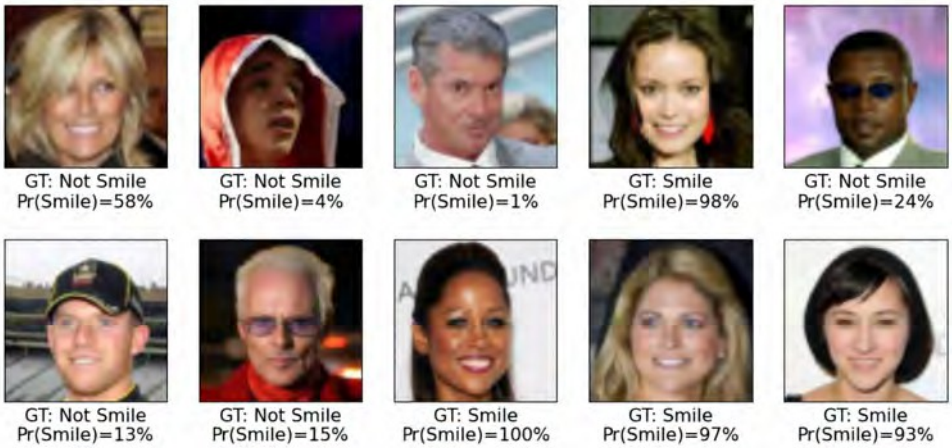
```
>> doğruluk_testi = 0
>> model.eval()
>> torch.no_grad() ile:
...     for x_batch, y_batch in test_dl:
...         pred = model(x_batch)[:, 0]
...         is_correct = ((pred>=0.5).float() == y_batch).float()
...         accuracy_test += is_correct.sum()
>>> accuracy_test /= len(test_dl.dataset)
>> print(f'Test doğruluğu: {accuracy_test:.4f}')
Test doğruluğu: 0,8446
```

Son olarak, bazı test örnekleri üzerinde tahmin sonuçlarını nasıl alacağımızı zaten biliyoruz. Aşağıdaki kodda, önceden işlenmiş test veri setimizin (test_dl) son partisinden 10 örneklik küçük bir alt küme alacağız. Ardından, her örneğin sınıf 1'den olma olasılıklarını hesaplayacağız (CelebA'da sağlanan etiketlere göre *gülümsemeye karşılık gelir*) ve örnekleri temel gerçek etiketleri ve tahmin edilen olasılıklarla birlikte görselleştireceğiz:

```
>>> pred = model(x_batch)[:, 0] * 100
>>> fig = plt.figure(figsize=(15, 7))
>> for j in range(10, 20):
...     ax = fig.add_subplot(2, 5, j-10+1)
...     ax.set_xticks([]); ax.set_yticks([])
...     ax.imshow(x_batch[j].permute(1, 2, 0))
...     if y_batch[j] == 1:
...         label='Gülümse'
...     başka:
...         label = 'Gülümseme'
...     ax.text(
...         0.5, -0.15,
...         f'GT: {label:s}\nPr(Smile)={pred[j]:.0f}%',
...         size=16,
...         horizontalalignment='center',
...         verticalalignment='center',
...         transform=ax.transAxes
...     )
>> plt.show()
```

 alinpikbarbell@gmail.com

*Şekil 14.20'*de, 10 örnek görüntüyü temel gerçek etiketleri ve sınıf 1, gülümseme sınıfına ait olma olasılıkları ile birlikte görebilirsiniz:



Şekil 14.20: Görüntü etiketleri ve bunların sınıf 1'e ait olma olasılıkları

Sınıf 1'in (yani CelebA'ya göre *gülümseme*) olasılıkları her görüntünün altında verilmiştir. Gördüğünüz gibi, eğitilmiş modelimiz 10 test örneğinden oluşan bu set üzerinde tamamen doğrudur.

İsteğe bağlı bir alıştırma olarak, oluşturduğumuz küçük alt küme yerine tüm eğitim veri kümesini kullanmayı denemeniz önerilir. Ayrıca, CNN mimarisini değiştirebilir veya modifiye edebilirsiniz. Örneğin, bırakma olasılıklarını ve farklı evrişimsel katmanlardaki filtre sayısını değiştirebilirsiniz. Ayrıca, küresel ortalama havuzlamayı tam bağlantılı bir katmanla değiştirebilirsiniz. Tüm eğitim veri setini bu bölümde eğittiğimiz CNN mimarisiyle kullanıyorsanız, yüzde 90'ın üzerinde doğruluk elde edebilmeniz gerekir.

Özet

Bu bölümde CNN'ler ve ana bileşenleri hakkında bilgi edindik. Konvolüsyon işlemi ile başladık ve 1D ve 2D uygulamalarını inceledik. Ardından, birçok yaygın CNN mimarisinde bulunan başka bir katman türünü ele aldık: alt örnekleme veya sözde havuzlama katmanları. Öncelikle en yaygın iki havuzlama biçimine odaklandık: maksimum havuzlama ve ortalama havuzlama.

Daha sonra, tüm bu bireysel kavramları bir araya getirerek, torch.nn modülünü kullanarak derin CNN'ler uyguladık. Uyguladığımız ilk ağ, zaten aşina olduğumuz MNIST el yazısı rakam tanıma problemine uygulandı.

Ardından, yüz görüntülerinden oluşan daha karmaşık bir veri kümesi üzerinde ikinci bir CNN uyguladık ve CNN'i gülümseme sınıflandırması için eğittik. Yol boyunca, torchvision.transforms modülünü kullanarak yüz görüntülerine uygulayabileceğimiz veri artırma ve farklı dönüşümler hakkında da bilgi edindiniz.

Bir sonraki bölümde, **tekrarlayan sinir ağlarına (RNN'ler)** geçeceğiz. RNN'ler dizi verilerinin yapısını öğrenmek için kullanılır ve dil çevirisi ve resim altyazısı gibi bazı büyüleyici uygulamaları vardır.

Kitabımızın Discord alanına katılın

Benzer düşünen insanlarla tanışmak ve 2000'den fazla üyemizle birlikte öğrenmek için Discord topluluğumuza katılın:
https://packt.link/MLwPyTorch

olimpikbarbell@gmail.com

15

Tekrarlayan Sinir Ağları Kullanarak Sıralı Verilerin Modellenmesi

Bir önceki bölümde, **evrişimli sinir ağlarına** (**CNN'ler**) odaklandık. CNN mimarilerinin yapı taşlarını ve PyTorch'ta derin CNN'lerin nasıl uygulanacağını ele aldık. Son olarak, görüntü sınıflandırması için CNN'lerin nasıl kullanılacağını öğrendiniz. Bu bölümde, **tekrarlayan sinir ağlarını** (**RNN'ler**) keşfedeceğiz ve sıralı verilerin modellenmesindeki uygulamalarını göreceğiz.

Aşağıdaki konuları ele alacağız:

- Sıralı verilerin tanıtılması
- Dizileri modellemek için RNN'ler
- Uzun kısa süreli bellek
- Zaman içinde kesilmiş geriye yayılım
- PyTorch'ta dizi modelleme için çok katmanlı bir RNN uygulama
- Birinci proje: IMDb film incelemesi veri kümesinin RNN duyarlılık analizi
- Proje iki: Jules Verne'in *Gizemli Ada'sından* metin verileri kullanarak LSTM hücreleri ile RNN karakter düzeyinde dil modelleme
- Degradelerin patlamasını önlemek için degrade kırpmayı kullanma

Sıralı verilerin tanıtılması

RNN'ler hakkındaki tartışmamıza, daha yaygın olarak dizi verileri veya **diziler** olarak bilinen sıralı verilerin doğasına bakarak başlayalım. Dizileri diğer veri türlerinden farklı kılan benzersiz özelliklerine bakacağız. Daha sonra sıralı verilerin nasıl temsil edileceğini göreceğiz ve bir modelin girdi ve çıktısına dayanan sıralı veriler için çeşitli model kategorilerini keşfedeceğiz. Bu, bu bölümde RNN'ler ve diziler arasındaki ilişkiyi keşfetmemize yardımcı olacaktır.

Sıralı verilerin modellenmesi - sıra önemlidir

Diğer veri türlerine kıyasla dizileri benzersiz kılan şey, bir dizideki öğelerin belirli bir sırayla ortaya çıkması ve birbirinden bağımsız olmamasıdır. Denetimli öğrenme için tipik makine öğrenimi algoritmaları, girdinin **bağımsız ve özdeş dağılımlı** (**IID**) veri olduğunu varsayar; bu da eğitim örneklerinin karşılıklı olarak *bağımsız olduğu ve* aynı temel **dağılıma** sahip olduğu anlamına gelir. Bu bağlamda, karşılıklı bağımsızlık varsayımına dayanarak, eğitim örneklerinin modele verilme sırası önemsizdir. Örneğin, $\boldsymbol{x}^{(1)}$ **,** $\boldsymbol{x}^{(2)}$, ..., $\boldsymbol{x}^{(n)}$ olmak üzere n eğitim örneğinden oluşan bir örneğimiz varsa, makine öğrenimi algoritmamızı eğitmek için verileri hangi sırada kullandığımız önemli değildir. Bu senaryoya bir örnek olarak daha önce çalıştığımız Iris veri kümesi verilebilir. Iris veri kümesinde her çiçek bağımsız olarak ölçülmüştür ve bir çiçeğin ölçümleri başka bir çiçeğin ölçümlerini etkilemez.

Ancak, bu varsayım dizilerle uğraştığımızda geçerli değildir - tanım gereği, sıra önemlidir. Belirli bir hisse senedinin piyasa değerini tahmin etmek bu senaryonun bir örneği olabilir. Örneğin, her bir eğitim örneğinin belirli bir gündeki belirli bir hisse senedinin piyasa değerini temsil ettiği n eğitim örneğinden oluşan bir örneğimiz olduğunu varsayalım. Görevimiz önümüzdeki üç gün için borsa değerini tahmin etmekse, bu eğitim örneklerini rastgele bir sırayla kullanmak yerine eğilimleri türetmek için önceki hisse senedi fiyatlarını tarih sıralı bir düzende dikkate almak mantıklı olacaktır.

Sıralı verilere karşı zaman serisi verileri

Zaman serisi verileri, her örneğin bir zaman boyutu ile ilişkilendirildiği özel bir sıralı veri türüdür. Zaman serisi verilerinde örnekler birbirini takip eden zaman damgalarında alınır ve bu nedenle zaman boyutu veri noktaları arasındaki sırayı belirler. Örneğin, hisse senedi fiyatları ve ses veya konuşma kayıtları zaman serisi verileridir.

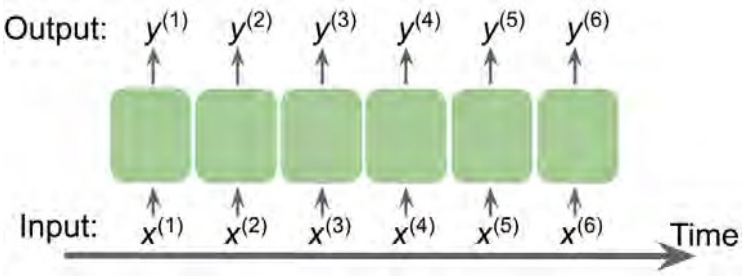
Öte yandan, tüm sıralı veriler zaman boyutuna sahip değildir. Örneğin, metin verilerinde veya DNA dizilerinde örnekler sıralıdır, ancak metin veya DNA zaman serisi verisi olarak nitelendirilmez. Göreceğiniz gibi, bu bölümde zaman serisi verisi olmayan doğal dil işleme (NLP) ve metin modelleme örneklerine odaklanacağız. Ancak, RNN'lerin zaman serisi verileri için de kullanılabileceğini ve bunun bu kitabın kapsamı dışında olduğunu unutmayın.

Dizileri temsil etme

Veri noktaları arasındaki sıralamanın sıralı verilerde önemli olduğunu tespit ettik, bu nedenle bir makine öğrenimi modelinde bu sıralama bilgisinden yararlanmanın bir yolunu bulmamız gerekiyor.

Bu bölüm boyunca dizileri $\langle \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(T)} \rangle$ şeklinde göstereceğiz. Üst simge indisleri örneklerin sırasını gösterir ve dizinin uzunluğu *T'dir. Dizilerin* mantıklı bir örneği için, her örnek noktasının, $x^{(t)}$, belirli bir zamana, *t,* ait olduğu zaman serisi verilerini düşünün. *Şekil 15.1,* hem giriş özelliklerinin (***x'ler) hem de*** hedef etiketlerin (***y'ler)*** doğal olarak zaman eksenlerine göre sırayı takip ettiği zaman serisi verilerinin bir örneğini göstermektedir; bu nedenle, hem ***x***'ler hem de ***y'ler*** dizilerdir.

alinpikbarbell@gmail.com



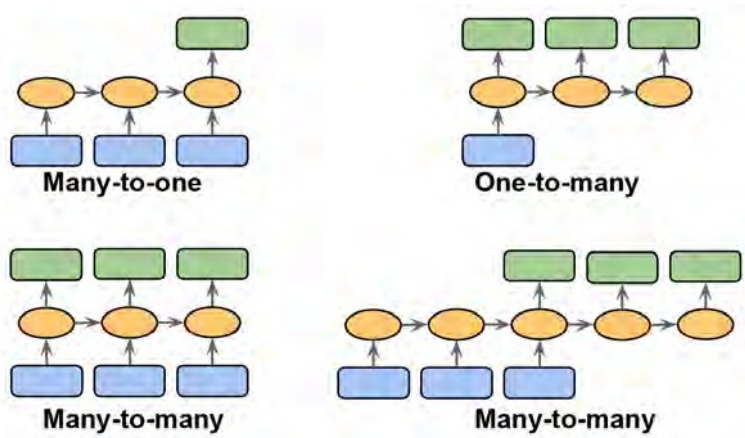
Şekil 15.1: Zaman serisi verilerine bir örnek

Daha önce de belirttiğimiz gibi, görüntü verileri için **çok katmanlı algılayıcılar** (**MLP'ler**) ve CNN'ler gibi şimdiye kadar ele aldığımız standart NN modelleri, eğitim örneklerinin birbirinden bağımsız olduğunu varsayar ve bu nedenle *sıralama bilgisini* içermez. Bu tür modellerin daha önce görülen eğitim örneklerine ilişkin bir *hafızaya* sahip olmadığını söyleyebiliriz. Örneğin, örnekler ileri besleme ve geri yayılma adımlarından geçirilir ve ağırlıklar eğitim örneklerinin işlenme sırasından bağımsız olarak güncellenir.

Buna karşın, RNN'ler dizileri modellemek için tasarlanmıştır ve geçmiş bilgileri hatırlama ve yeni olayları buna göre işleme yeteneğine sahiptir, bu da dizi verileriyle çalışırken açık bir avantajdır.

Farklı sekans modelleme kategorileri

Sıra modelleme, dil çevirisi (örneğin, İngilizce'den Almanca'ya metin çevirisi), görüntü altyazısı ve metin oluşturma gibi birçok büyüleyici uygulamaya sahiptir. Ancak, uygun bir mimari ve yaklaşım seçebilmek için bu farklı dizi modelleme görevlerini anlamamız ve birbirinden ayırt edebilmemiz gerekir. *Andrej Karpathy,* 2015 (http:// karpathy.github.io/2015/05/21/rnn-effectiveness/) tarafından yazılan *The Unreasonable* Effectiveness *of Recurrent Neural Networks adlı* mükemmel makaledeki açıklamalara dayanan *Şekil 15.2,* girdi ve çıktı verilerinin ilişki kategorilerine bağlı olan en yaygın dizi modelleme görevlerini özetlemektedir.



Şekil 15.2: En yaygın sıralama görevleri

Bir önceki şekilde tasvir edilen girdi ve çıktı verileri arasındaki farklı ilişki kategorilerini daha ayrıntılı olarak tartışalım. Eğer ne girdi ne de çıktı verileri dizileri temsil etmiyorsa, o zaman standart verilerle uğraşıyoruz demektir ve bu tür verileri modellemek için basitçe çok katmanlı bir algılayıcı (ya da bu kitapta daha önce ele alınan başka bir sınıflandırma modeli) kullanabiliriz. Ancak, girdi ya da çıktı bir diziyse, modelleme görevi muhtemelen bu kategorilerden birine girer:

- **Çoktan bire**: Giriş verileri bir dizidir, ancak çıktı bir dizi değil, sabit boyutlu bir vektör veya skalerdir. Örneğin, duyarlılık analizinde girdi metin tabanlıdır (örneğin, bir film eleştirisi) ve çıktı bir sınıf etiketidir (örneğin, bir eleştirmenin filmi beğenip beğenmediğini gösteren bir etiket).
- **Birden çoğa**: Giriş verileri standart formattadır ve bir dizi değildir, ancak çıktı bir dizidir. Bu kategoriye örnek olarak resim altyazısı verilebilir; girdi bir resimdir ve çıktı da bu resmin içeriğini özetleyen İngilizce bir ifadedir.
- **Çoktan çoğa**: Hem giriş hem de çıkış dizileri dizilerdir. Bu kategori, giriş ve çıkışın senkronize edilip edilmediğine göre daha da bölünebilir. Senkronize edilmiş çoktan çoğa modelleme görevine örnek olarak videodaki her karenin etiketlendiği video sınıflandırma verilebilir. *Gecikmeli çoktan* çoğa modelleme görevine örnek olarak bir dili başka bir dile çevirmek verilebilir. Örneğin, İngilizce bir cümlenin Almanca'ya çevirisi üretilmeden önce tamamının bir makine tarafından okunması ve işlenmesi gerekir.

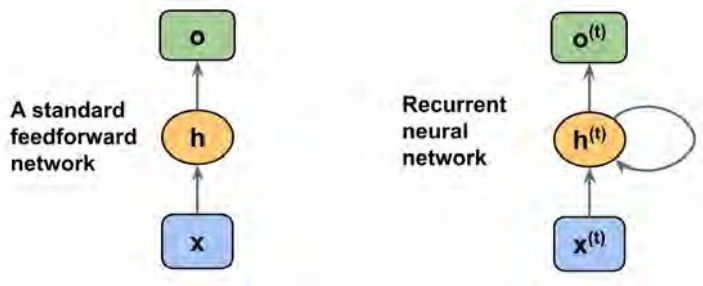
Şimdi, üç geniş dizi modelleme kategorisini özetledikten sonra, bir RNN'nin yapısını tartışmaya geçebiliriz.

Dizileri modellemek için RNN'ler

Bu bölümde, PyTorch'ta RNN'leri uygulamaya başlamadan önce, RNN'lerin ana kavramlarını tartışacağız. Sıra verilerini modellemek için özyinelemeli bir bileşen içeren bir RNN'nin tipik yapısına bakarak başlayacağız. Ardından, nöron aktivasyonlarının tipik bir RNN'de nasıl hesaplandığını inceleyeceğiz. Bu, RNN'lerin eğitimindeki yaygın zorlukları tartışmamız için bir bağlam oluşturacak ve daha sonra LSTM ve **kapılı tekrarlayan birimler** (**GRU'lar**) gibi bu zorluklara yönelik çözümleri tartışacağız.

RNN'lerde veri akışını anlama

Bir RNN mimarisi ile başlayalım. *Şekil 15.3* standart bir ileri beslemeli NN'deki ve bir RNN'deki veri akışını karşılaştırma için yan yana göstermektedir:



Şekil 15.3: Standart bir ileri beslemeli NN ve bir RNN'nin veri akışı

 olimpikbarbell@gmail.com

Bu ağların her ikisi de yalnızca bir gizli katmana sahiptir. Bu gösterimde birimler gösterilmemektedir, ancak giriş katmanının (***x***), gizli katmanın (***h***) ve çıkış katmanının (***o***) birçok birim içeren vektörler olduğunu varsayıyoruz.

Bir RNN'den elde edilen çıktı türünü belirleme

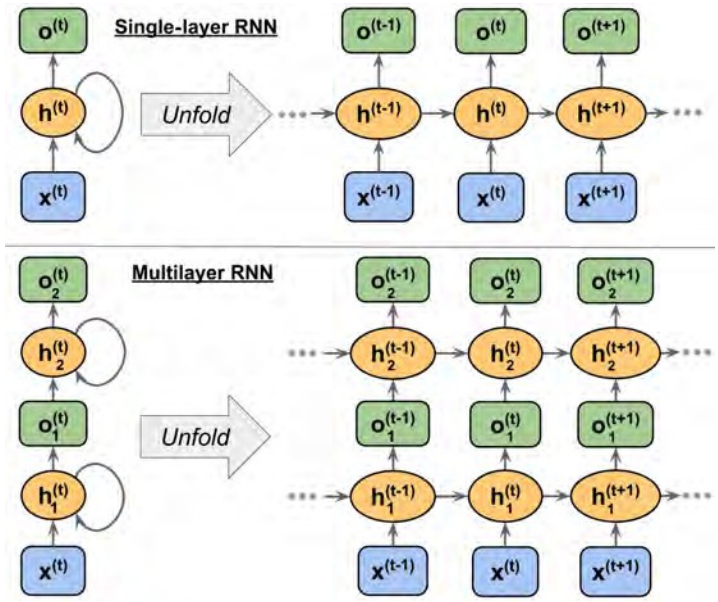
Bu genel RNN mimarisi, girdinin bir dizi olduğu iki dizi modelleme kategorisine karşılık gelebilir. Tipik olarak, tekrarlayan bir katman çıktı olarak bir dizi döndürebilir, $\langle o^{(0)}, o^{(1)}, \ldots, o^{(T)} \rangle$ veya sadece son çıktıyı döndürebilir ($t = T$'de, yani $\boldsymbol{o}^{(T)}$). Böylece, çoktan çoğa olabileceği gibi, örneğin son çıktı olarak sadece son elemanı, $\boldsymbol{o}^{(T)}$, kullanırsak çoktan bire de olabilir.

Bunun PyTorch torch.nn modülünde nasıl ele alındığını daha sonra, bir diziyi çıktı olarak döndürmekle ilgili olarak tekrarlayan bir katmanın davranışına kuyruklu bir bakış attığımızda göreceğiz.

Standart bir ileri beslemeli ağda bilgi girdiden gizli katmana ve ardından gizli katmandan çıktı katmanına akar. Öte yandan, bir RNN'de gizli katman, girdisini hem mevcut zaman adımının girdi katmanından hem de bir önceki zaman adımının gizli katmanından alır.

Gizli katmandaki bitişik zaman adımlarındaki bilgi akışı, ağın geçmiş olaylara dair bir hafızaya sahip olmasını sağlar. Bu bilgi akışı genellikle bir döngü olarak gösterilir, grafik gösteriminde **tekrarlayan kenar** olarak da bilinir ve bu genel RNN mimarisi adını bu şekilde almıştır.

Çok katmanlı algılayıcılara benzer şekilde, RNN'ler birden fazla gizli katmandan oluşabilir. *Tek* gizli katmanlı RNN'lere tek katmanlı *RNN* olarak atıfta bulunmanın yaygın bir gelenek olduğunu ve bunun Adaline veya lojistik regresyon gibi gizli katmanı olmayan tek katmanlı NN'lerle karıştırılmaması gerektiğini unutmayın. *Şekil 15.4'te* bir gizli katmana sahip bir RNN (üstte) ve iki gizli katmana sahip bir RNN (altta) gösterilmektedir:



6587191CA584E

Şekil 15.4: Bir ve iki gizli katmanlı RNN örnekleri

olimpikbarbell@gmail.com

RNN'lerin mimarisini ve bilgi akışını incelemek için, *Şekil 15.4'te* görebileceğiniz tekrarlayan kenarlı kompakt bir gösterim açılabilir.

Bildiğimiz gibi, standart bir NN'deki her gizli birim yalnızca bir girdi alır - girdi katmanıyla ilişkili net ön aktivasyon. Bunun aksine, bir RNN'deki her gizli birim iki *farklı* girdi seti alır - giriş katmanından gelen ön aktivasyon ve aynı gizli katmanın bir önceki zaman adımındaki aktivasyonu, t - 1.

İlk zaman adımında, $t = 0$, gizli birimler sıfıra veya küçük rastgele değerlere başlatılır. Daha sonra, $t > 0$ olduğu bir zaman adımında, gizli birimler girdilerini mevcut zamandaki veri noktasından, $\boldsymbol{x}^{(t)}$, ve gizli birimlerin t - 1'deki önceki değerlerinden, $\boldsymbol{h}^{(t-1)}$ olarak gösterilir.

Benzer şekilde, çok katmanlı bir RNN söz konusu olduğunda, bilgi akışını aşağıdaki gibi özetleyebiliriz:

- *katman* = 1: Burada, gizli katman $h^{(t)}_1$ olarak temsil edilir ve girdisini veri noktasından, $\boldsymbol{x}^{(t)}$, ve aynı katmandaki gizli değerlerden alır, ancak bir önceki zaman adımında, $h^{(t-1)}_1$.
- *katman* = 2: İkinci gizli katman, $h^{(t)}_2$, girişlerini aşağıdaki katmanın çıkışlarından alır mevcut zaman adımındaki ($o^{(t)}_1$) ve bir önceki zaman adımındaki kendi gizli değerleri, $h^{(t-1)}_2$.

Bu durumda, her bir tekrarlayan katmanın girdi olarak bir dizi alması gerektiğinden, sonuncusu hariç tüm tekrarlayan katmanlar *çıktı olarak bir dizi döndürmelidir* (yani, daha sonra return_sequences=True *olarak* ayarlamamız gerekecektir). Son tekrarlayan katmanın davranışı problemin türüne bağlıdır.

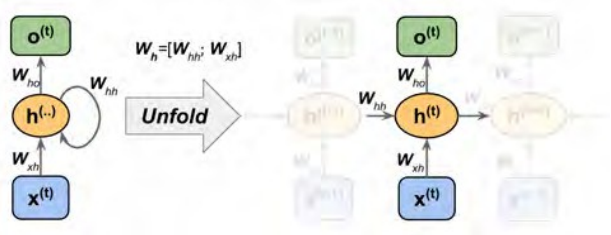
Bir RNN'de aktivasyonların hesaplanması

Artık bir RNN'deki yapıyı ve genel bilgi akışını anladığınıza göre, daha spesifik olalım ve gizli katmanların yanı sıra çıktı katmanının gerçek aktivasyonlarını hesaplayalım. Basitlik açısından, sadece tek bir gizli katmanı ele alacağız; ancak aynı kavram çok katmanlı RNN'ler için de geçerlidir.

Az önce incelediğimiz RNN temsilindeki her bir yönlendirilmiş kenar (kutular arasındaki bağlantılar) bir ağırlık matrisi ile ilişkilidir. Bu ağırlıklar zamana (t) bağlı değildir; dolayısıyla zaman ekseni boyunca paylaşılırlar. Tek katmanlı bir RNN'deki farklı ağırlık matrisleri aşağıdaki gibidir:

- $\boldsymbol{W}_{xh}$: Giriş, $\boldsymbol{x}^{(t)}$ ve gizli katman, $\boldsymbol{h}$ arasındaki ağırlık matrisi
- $\boldsymbol{W}_{hh}$: Tekrarlayan kenar ile ilişkili ağırlık matrisi
- $\boldsymbol{W}_{ho}$: Gizli katman ve çıktı katmanı arasındaki ağırlık matrisi Bu

ağırlık matrisleri *Şekil 15.5*'te gösterilmiştir:



Şekil 15.5: Tek katmanlı bir RNN'ye ağırlıkların uygulanması

Bazı uygulamalarda, ağırlık matrislerinin, $\boldsymbol{W}_{xh}$ ve $\boldsymbol{W}$, $\boldsymbol{W}_{hhh} = [\boldsymbol{W}_{xh}\ ;\ \boldsymbol{W}_{hh}\ ]$ birleşik matrisine birleştirildiğini gözlemleyebilirsiniz. Bu bölümün ilerleyen kısımlarında bu gösterimi de kullanacağız.

Aktivasyonların hesaplanması standart çok katmanlı algılayıcılara ve diğer ileri beslemeli NN türlerine çok benzer. Gizli katman için net girdi, z_h (ön aktivasyon), doğrusal bir kombinasyon yoluyla hesaplanır; yani, ağırlık matrislerinin ilgili vektörlerle çarpımlarının toplamını hesaplar ve önyargı birimini ekleriz:

$$z_h^{(t)} = W_{xh} x^{(t)} + W_{hh} h^{(t-1)} + b_h$$

Daha sonra, t zaman adımında gizli birimlerin aktivasyonları aşağıdaki gibi hesaplanır:

$$h_h^{(t)} = \sigma_h(z_h^{(t)}) = \sigma_h(W_{xh} x^{(t)} + W_{hh} h^{(t-1)} + b_h)$$

Burada, $\boldsymbol{b}_h$ gizli birimler için bias vektörü ve σ gizli katmanın aktivasyon fonksiyonudur.

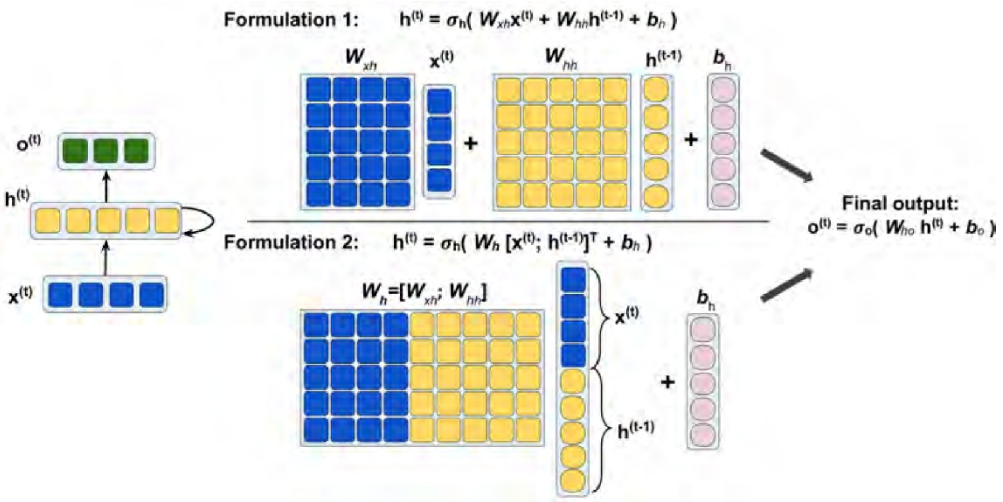
Birleştirilmiş ağırlık matrisini kullanmak istemeniz durumunda, $\boldsymbol{W}_h = [\boldsymbol{W}_{xh}\ ;\ \boldsymbol{W}_{hh}\ ]$, gizli birimleri hesaplama formülü aşağıdaki gibi değişecektir:

$$h_h^{(t)} = \sigma_h([W_{xh}; W_{hh}] \begin{bmatrix} x^{(t)} \\ h^{(t-1)} \end{bmatrix} + b_h)$$

Gizli birimlerin mevcut zaman adımındaki aktivasyonları hesaplandıktan sonra, çıkış birimlerinin aktivasyonları aşağıdaki gibi hesaplanacaktır:

$$o_o^{(t)} = \sigma_o(W_{ho} h_h^{(t)} + b_o)$$

Bunu daha da açıklığa kavuşturmak için *Şekil 15.6'da* bu aktivasyonların her iki formülasyonla hesaplanma süreci gösterilmektedir:



Şekil 15.6: Aktivasyonların hesaplanması

 olimpikbarbell@gmail.com

Zaman içinde geri yayılım (BPTT) kullanarak RNN'leri eğitme

RNN'ler için öğrenme algoritması 1990 yılında tanıtılmıştır: *Backpropagation Through Time: What It Does and How to Do It* (*Paul Werbos*, *Proceedings of IEEE*, 78(10): 1550-1560, 1990).

Gradyanların türetilmesi biraz karmaşık olabilir, ancak temel fikir, genel kaybın, L, $t = 1$ ila $t = T$ zamanlarındaki tüm kayıp fonksiyonlarının toplamı olduğudur:

$$L = \sum_{t=1}^{T} L^{(t)}$$

t zamanındaki kayıp, önceki tüm zaman adımları 1 : t'deki gizli birimlere bağlı olduğundan, gradyan aşağıdaki gibi hesaplanacaktır:

$$\frac{\partial L^{(t)}}{\partial \theta_{hh}} = \frac{\partial L^{(t)}}{\partial o^{(t)}} \times \frac{\partial o^{(t)}}{\partial h^{(t)}} \times \left(\sum_{k=1}^{t} \frac{\partial h^{(t)}}{\partial h^{(k)}} \times \frac{\partial h^{(k)}}{\partial \theta_{hh}} \right)$$

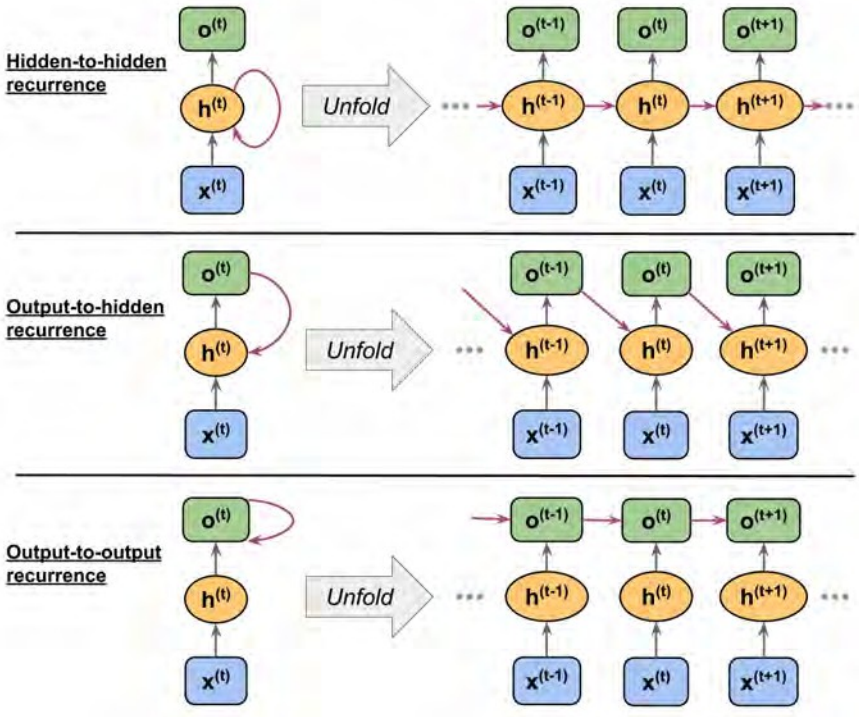
İşte, $\frac{\partial h^{(t)}}{\partial h^{(k)}}$ bitişik zaman adımlarının çarpımı olarak hesaplanır:

$$\frac{\partial h^{(t)}}{\partial h^{(k)}} = \prod_{i=k+1}^{t} \frac{\partial h^{(i)}}{\partial h^{(i-1)}}$$

Gizli yinelemeye karşı çıktı yinelemesi

Şimdiye kadar, gizli katmanın tekrarlayan özelliğe sahip olduğu tekrarlayan ağları gördünüz. Bununla birlikte, tekrarlayan bağlantının çıktı katmanından geldiği alternatif bir model olduğunu unutmayın. Bu durumda, bir önceki zaman adımında çıktı katmanından gelen net aktivasyonlar, $\boldsymbol{o}^{t-1}$, iki yoldan biriyle eklenebilir:

- Geçerli zaman adımındaki gizli katmana, $\boldsymbol{h}^t$ (*Şekil 15.7*'de çıktı-gizli yineleme olarak gösterilmiştir)
- Geçerli zaman adımındaki çıktı katmanına, $\boldsymbol{o}^t$ (*Şekil 15.7*'de çıktıdan çıktıya yeniden akım olarak gösterilmiştir)



Şekil 15.7: Farklı tekrarlayan bağlantı modelleri

*Şekil 15.7'*de gösterildiği gibi, bu mimariler arasındaki farklar yinelenen halka bağlantılarında açıkça görülebilir. Notasyonumuzu takip ederek, tekrarlayan bağlantı ile ilişkili ağırlıklar gizli-gizli tekrarlama için $\boldsymbol{W}_{hh}$ ***, çıkış-gizli tekrarlama için*** $\boldsymbol{W}_{oh}$ ve çıkış-çıkış tekrarlama için $\boldsymbol{W}_{oo}$ ile gösterilecektir. Literatürdeki bazı makalelerde, tekrarlayan bağlantılarla ilişkili ağırlıklar $\boldsymbol{W}_{rec}$ ile de gösterilmektedir.

Bunun pratikte nasıl çalıştığını görmek için, bu tekrarlayan tiplerden biri için ileri geçişi manuel olarak hesaplayalım. torch.nn modülü kullanılarak, gizli-gizli yinelemeye benzer şekilde RNN aracılığıyla bir yinelemeli katman tanımlanabilir. Aşağıdaki kodda, RNN'den bir tekrarlayan katman oluşturacağız ve çıktıyı hesaplamak için 3 uzunluğundaki bir giriş dizisi üzerinde bir ileri geçiş gerçekleştireceğiz. Ayrıca ileri geçişi manuel olarak hesaplayacak ve sonuçları RNN'ninkilerle karşılaştıracağız.

 olimpikbarbell@gmail.com

İlk olarak, katmanı oluşturalım ve manuel hesaplamalarımız için ağırlıkları ve önyargıları atayalım:

```
>> import torch
>> import torch.nn as nn
>>> torch.manual_seed(1)
>> rnn_layer = nn.RNN(input_size=5, hidden_size=2,
...                   num_layers=1, batch_first=True)
>> w_xh = rnn_layer.weight_ih_l0
>> w_hh = rnn_layer.weight_hh_l0
>>> b_xh = rnn_layer.bias_ih_l0
>>> b_hh = rnn_layer.bias_hh_l0
>> print('W_xh shape:', w_xh.shape)
>> print('W_hh shape:', w_hh.shape)
>> print('b_xh shape:', b_xh.shape)
>> print('b_hh şekli:', b_hh.shape)
W_xh şekli: torch.Size([2, 5])
W_hh şekli: torch.Size([2, 2])
b_xh şekli: torch.Size([2])
b_hh şekli: torch.Size([2])
```

Bu katman için girdi şekli (batch_size, sequence_length, 5) şeklindedir; burada ilk boyut yığın boyutudur (batch_first=True olarak ayarladığımız için), ikinci boyut diziye karşılık gelir ve son boyut özelliklere karşılık gelir. Uzunluğu 3 olan bir girdi dizisi için $\langle o^{(0)}, o^{(1)}, o^{(2)} \rangle$ çıktı dizisiyle sonuçlanacak bir dizi çıktısı alacağımıza dikkat edin. Ayrıca, RNN varsayılan olarak bir katman kullanır ve yığılmış bir RNN oluşturmak üzere birden fazla RNN katmanını bir araya getirmek için num_layers ayarını yapabilirsiniz.

Şimdi, rnn_layer üzerinde ileri geçişi çağıracağız ve her zaman adımında çıktıları manuel olarak hesaplayıp karşılaştıracağız:

```
>> x_seq = torch.tensor([[1.0]*5, [2.0]*5, [3.0]*5]).float()
>> ## basit RNN'nin çıktısı:
>> output, hn = rnn_layer(torch.reshape(x_seq, (1, 3, 5))
>> ## çıktıyı manuel olarak hesaplar:
>>> out_man = []
>> for t in range(3):
...     xt = torch.reshape(x_seq[t], (1, 5))
...     print(f'Zaman adımı {t} =>')
...     print('    Giriş        :', xt.numpy())
...
...     ht = torch.matmul(xt, torch.transpose(w_xh, 0, 1)) + b_xh
...     print('    Gizli        :', ht.detach().numpy())
...
```

```
...     t > 0 ise:
...         prev_h = out_man[t-1]
...     Başka:
...         prev_h = torch.zeros((ht.shape))
...     ot = ht + torch.matmul(prev_h, torch.transpose(w_hh, 0, 1)) \
...             + b_hh
...     ot = torch.tanh(ot) out_man.append(ot)
...
...     print('  Çıktı (manuel) :', ot.detach().numpy())
...     print('  RNN çıktısı      :', output[:, t].detach().numpy())
...     print()
Zaman adımı 0
=>
    Giriş              : [[1. 1. 1. 1.  1.]]
    Gizli              : [[-0.4701929  0.5863904]]
     Çıkış (manuel) :  [[-0.3519801    0.52525216]]
    RNN çıktısı        : [[-0.3519801    0.52525216]]

Zaman adımı 1
=>
    Giriş              :  [[2. 2. 2. 2.  2.]]
    Gizli              : [[-0.88883156   1.2364397 ]]
     Çıkış (manuel) :  [[-0.68424344   0.76074266]]
    RNN çıktısı        : [[-0.68424344   0.76074266]]

Zaman adımı 2
=>
    Giriş              : [[3. 3. 3. 3.  3.]]
    Gizli              : [[-1.3074701  1.886489 ]]
     Çıkış (manuel) :  [[-0.8649416    0.90466356]]
    RNN çıktısı        : [[-0.8649416    0.90466356]]
```

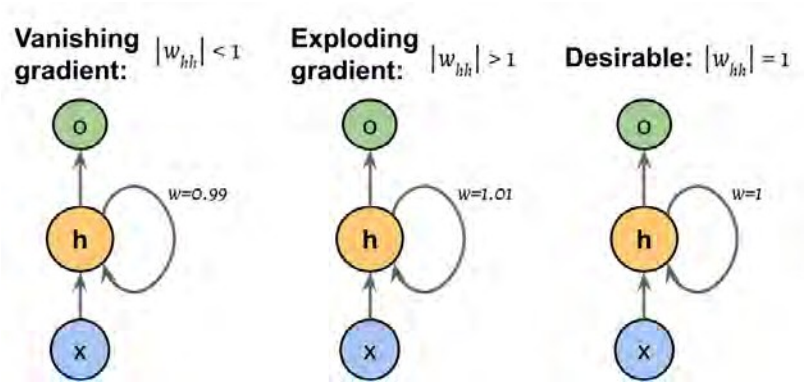
Manuel ileri hesaplamamızda, RNN'de de kullanıldığı için hiperbolik tanjant (tanh) aktivasyon fonksiyonunu kullandık (varsayılan aktivasyon). Yazdırılan sonuçlardan da görebileceğiniz gibi, manuel ileri hesaplamalardan elde edilen çıktılar, her zaman adımında RNN katmanının çıktısıyla tam olarak eşleşmektedir. Umarım, bu uygulamalı görev sizi tekrarlayan ağların gizemleri konusunda aydınlatmıştır.

Uzun menzilli etkileşimleri öğrenmenin zorlukları

Daha önce kısaca bahsedilen BPTT, bazı yeni zorlukları da beraberinde getirmektedir. Çünkü çoklu- $\frac{\partial h^{(t)}}{\partial h^{(k)}}$, bir kayıp fonksiyonunun gradyanlarının hesaplanmasında, **kaybolan** ve **patlayan** olarak adlandırılan gradyan sorunları ortaya çıkar.

alinpikbarbell@gmail.com

Bu sorunlar, basitlik için sadece bir gizli birime sahip bir RNN'yi gösteren *Şekil 15.8'*deki örneklerle açıklanmaktadır:



Şekil 15.8: Kayıp fonksiyonunun gradyanlarının hesaplanmasındaki sorunlar

Temel olarak, $\frac{\partial h^{(t)}}{\partial h^{(k)}}$ *t - k* çarpımına sahiptir; bu nedenle, ağırlık, *w,* kendisi ile *t - k* kez çarpıldığında w^{t-k}Sonuç olarak, eğer |w| < 1 ise, *t - k* büyük olduğunda bu faktör çok küçük olur. Öte yandan, tekrarlayan kenarın ağırlığı |w| > 1 ise, *t - k büyük* olduğunda w^{t-k} çok büyük olur. Büyük bir *t - k değerinin* uzun menzilli bağımlılıkları ifade ettiğini unutmayın. Kaybolan veya patlayan gradyanlardan kaçınmak için naif bir çözüme |*w*| = 1 sağlanarak ulaşılabileceğini görebiliriz. Eğer ilgileniyorsanız ve bu konuyu daha detaylı incelemek isterseniz, *R. Pascanu, T. Mikolov* ve *Y. Bengio'nun On the difficulty of training recurrent neural networks,* 2012 (https://arxiv.org/pdf/1211.5063.pdf) adlı kitabını okuyabilirsiniz.

Uygulamada bu sorunun en az üç çözümü vardır:

- Degrade kırpma
- **Zaman içinde kesilmiş geri yayılım** (**TBPTT**)
- LSTM

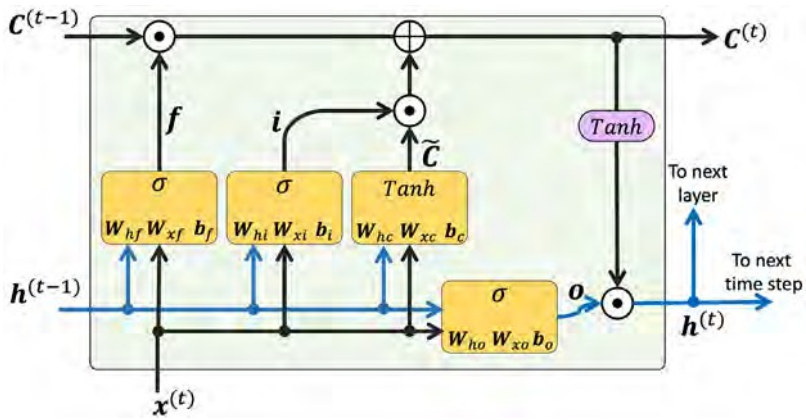
Gradyan kırpmayı kullanarak, gradyanlar için bir kesme veya eşik değeri belirleriz ve bu değeri aşan gradyan değerlerine bu kesme değerini atarız. Buna karşılık TBPTT, sinyalin her ileri geçişten sonra geri yayılabileceği zaman adımı sayısını sınırlar. Örneğin, dizinin 100 öğesi veya adımı olsa bile, yalnızca en son 20 zaman adımını geri yayabiliriz.

Hem gradyan kırpma hem de TBPTT patlayan gradyan sorununu çözebilirken, kırpma, gradyanın etkili bir şekilde geri akabileceği ve ağırlıkları düzgün bir şekilde güncelleyebileceği adım sayısını sınırlar. Öte yandan, 1997 yılında Sepp Hochreiter ve Jürgen Schmidhuber tarafından tasarlanan LSTM, bellek hücrelerinin kullanımı yoluyla uzun menzilli bağımlılıkları modellerken kaybolan ve patlayan gradyan problemlerinde daha başarılı olmuştur. LSTM'yi daha ayrıntılı olarak tartışalım.

Uzun kısa süreli bellek hücreleri

Daha önce de belirtildiği gibi, LSTM'ler ilk olarak kaybolan gradyan probleminin üstesinden gelmek için tanıtılmıştır (*Long Short-Term Memory, S. Hochreiter* ve *J. Schmidhuber*, *Neural Computation,* 9(8): 1735-1780, 1997). Bir LSTM'nin yapı taşı, esasen standart RNN'lerin gizli katmanını temsil eden veya onun yerini alan bir **bellek hücresidir**.

Her bellek hücresinde, kaybolan ve patlayan gradyan sorunlarının üstesinden gelmek için tartıştığımız gibi $w = 1$ arzu edilen ağırlığa sahip bir tekrarlayan kenar vardır. Bu tekrarlayan kenarla ilişkili değerler topluca **hücre durumu** olarak adlandırılır. Modern bir LSTM hücresinin katlanmamış yapısı *Şekil 15.9*'da gösterilmektedir:



Şekil 15.9: Bir LSTM hücresinin yapısı

Bir önceki zaman adımındaki hücre durumunun, $\boldsymbol{C}^{(t-1)}$, doğrudan herhangi bir ağırlık faktörü ile çarpılmadan, mevcut zaman adımındaki hücre durumunu, $\boldsymbol{C}^{(t)}$, elde etmek için değiştirildiğine dikkat edin. Bu bellek hücresindeki bilgi akışı, burada açıklanacak olan birkaç hesaplama birimi (genellikle *kapı olarak* adlandırılır) tarafından kontrol edilir. Şekilde, $\odot$ **eleman-bilge** çarpımı (eleman-bilge çarpma) ve $\oplus$ **eleman-bilge toplama** (eleman-bilge toplama) anlamına gelmektedir. Ayrıca, $\boldsymbol{x}^{(t)}$ t zamanındaki girdi verilerini ve $\boldsymbol{h}^{(t-1)}$ t - 1 zamanındaki gizli birimleri ifade eder. Dört kutu, sigmoid fonksiyonu (σ) veya tanh gibi bir aktivasyon fonksiyonu ve bir dizi ağırlık ile gösterilir; bu kutular, girişleri üzerinde matris-vektör çarpımları gerçekleştirerek doğrusal bir kombinasyon uygular (bunlar $\boldsymbol{h}^{(t-1)}$ ve $\boldsymbol{x}^{(t)}$). Çıkış birimleri $\odot$'den geçirilen sigmoid aktivasyon fonksiyonlarına sahip bu birleştirme birimlerine geçit adı verilir.

Bir LSTM hücresinde, unutma kapısı, giriş kapısı ve çıkış kapısı olarak bilinen üç farklı kapı türü vardır:

Unutma kapısı ($\boldsymbol{f}_t$) bellek hücresinin sonsuza kadar büyümeden hücre durumunu sıfırlamasını sağlar. Aslında unutma kapısı hangi bilginin geçmesine izin verileceğine ve hangi bilginin bastırılacağına karar verir. Şimdi, $\boldsymbol{f}_t$ aşağıdaki gibi hesaplanır:

$$_{ft} = \sigma\sigma\sigma xf^{xx(t)} + \sigma\sigma hf^{h(tt1)} + {}_{bf})$$

 olimpikbarbell@gmail.com

Unutma kapısının orijinal LSTM hücresinin bir parçası olmadığına dikkat ediniz; orijinal modeli geliştirmek için birkaç yıl sonra eklenmiştir (*Learning to Forget: LSTM ile Sürekli Tahmin*, *F. Gers*, *J. Schmidhuber* ve *F. Cummins*, *Neural Computation 12,* 2451-2471, 2000).

Giriş kapısı (i_t) ve **aday değer** ($\tilde{C}_t$) hücre durumunun güncellenmesinden sorumludur. Bunlar aşağıdaki gibi bir araya getirilir:

$$i_t = \sigma(W_{xi}x_{(t)} + W_{hi}h_{(t-1)} + b_i)$$

$$\tilde{C}_t = \tanh(W_{xc}x_{(t)} + W_{hc}h_{(t-1)} + b_c)$$

t zamanındaki hücre durumu aşağıdaki gibi hesaplanır:

$$C_{(t)} = (C_{(t-1)} \odot f_t) \oplus (i_t \odot \tilde{C}_t)$$

Çıkış kapısı (o_t) gizli birimlerin değerlerinin nasıl güncelleneceğine karar verir:

$$o_t = \sigma(W_{xo}x_{(t)} + W_{ho}h_{(t-1)} + b_o)$$

Buna göre, mevcut zaman adımındaki gizli birimler aşağıdaki gibi hesaplanır:

$$h_{(t)} = o_t \odot \tanh(C_{(t)})$$

Bir LSTM hücresinin yapısı ve altında yatan hesaplamalar çok karmaşık ve uygulaması zor görünebilir. Ancak iyi haber şu ki PyTorch, LSTM hücrelerimizi kolay ve verimli bir şekilde tanımlamamıza olanak tanıyan optimize edilmiş sarmalayıcı işlevlerde her şeyi zaten uygulamıştır. RNN'leri ve LSTM'leri bu bölümün ilerleyen kısımlarında gerçek dünya veri kümelerine uygulayacağız.

Diğer gelişmiş RNN modelleri

LSTM'ler, dizilerdeki uzun menzilli bağımlılıkları modellemek için temel bir yaklaşım sağlar. Ancak, literatürde LSTM'lerin birçok varyasyonunun tanımlandığını belirtmek önemlidir (*An Empirical Exploration of Recurrent Network Architectures* by *Rafal Jozefowicz*, *Wojciech Zaremba*, and *Ilya Sutskever*, *Proceedings of ICML,* 2342-2350, 2015). Ayrıca, 2014 yılında önerilen daha yeni bir yaklaşım olan **kapılı tekrarlayan birim** (**GRU**) de kayda değerdir. GRU'lar LSTM'lerden daha basit bir mimariye sahiptir; bu nedenle, hesaplama açısından daha verimli olmakla birlikte, polifonik müzik modelleme gibi bazı görevlerdeki performansları LSTM'lerle karşılaştırılabilir. Bu modern RNN mimarileri hakkında daha fazla bilgi edinmek istiyorsanız, *Junyoung Chung* ve diğerleri tarafından 2014 yılında yayınlanan *Gated Recurrent Neural Networks on Sequence Modeling adlı* makaleye bakabilirsiniz (https://arxiv.org/pdf/1412.3555v1.pdf).

PyTorch'ta dizi modelleme için RNN'lerin uygulanması

Artık RNN'lerin altında yatan teoriyi ele aldığımıza göre, bu bölümün daha pratik kısmına geçmeye hazırız: RNN'leri PyTorch'ta uygulamak. Bu bölümün geri kalanında, RNN'leri iki yaygın problem görevine uygulayacağız:

1. Duygu analizi
2. Dil modelleme

İlerleyen sayfalarda birlikte inceleyeceğimiz bu iki proje hem büyüleyici hem de oldukça karmaşıktır. Bu nedenle, kodu tek seferde sunmak yerine, uygulamayı birkaç adıma bölecek ve kodu ayrıntılı olarak tartışacağız. Büyük resme genel bir bakış atmak ve tartışmaya dalmadan önce tüm kodu bir kerede görmek istiyorsanız, önce kod uygulamasına bir göz atın.

Birinci proje - IMDb film incelemelerinin duyarlılığını tahmin etme

Duygu analizinin bir cümle veya metin belgesinin ifade edilen görüşünü analiz etmekle ilgili olduğunu *Bölüm 8*, Duygu *Analizine Makine Öğrenimini Uygulama'*dan hatırlayabilirsiniz. Bu bölümde ve sonraki alt bölümlerde, duygu analizi için çok katmanlı bir RNN'yi çoktan bire bir mimari kullanarak uygulayacağız.

Bir sonraki bölümde, bir dil modelleme uygulaması için çoktan çoğa bir RNN uygulayacağız. Seçilen örnekler, RNN'lerin ana kavramlarını tanıtmak için kasıtlı olarak basit olsa da, dil modellemenin, bilgisayarlara insanlarla doğrudan konuşma ve etkileşim kurma yeteneği veren sohbet robotları oluşturmak gibi çok çeşitli ilginç uygulamaları vardır.

Film inceleme verilerinin hazırlanması

Bölüm 8'de, inceleme veri setini önceden işledik ve temizledik. Şimdi de aynısını yapacağız. İlk olarak, gerekli modülleri içe aktaracağız ve verileri torchtext'ten (pip install torchtext aracılığıyla kuracağız; 2021'in sonlarında 0.10.0 sürümü kullanılıyordu) aşağıdaki gibi

```
>> from torchtext.datasets import IMDB
>> train_dataset = IMDB(split='train')
>> test_dataset = IMDB(split='test')
```

okuyacağız:

Her sette 25.000 örnek bulunmaktadır. Veri kümelerinin her bir örneği iki unsurdan oluşur: tahmin etmek istediğimiz hedef etiketi temsil eden duygu etiketi (neg olumsuz duyguyu ve pos olumlu duyguyu ifade eder) ve film inceleme metni (girdi özellikleri). Bu film incelemelerinin metin bileşeni kelime dizileridir ve RNN modeli her bir diziyi pozitif (1) veya negatif (0) inceleme olarak sınıflandırır.

 olimpikbarbell@gmail.com

Bununla birlikte, verileri bir RNN modeline beslemeden önce, birkaç ön işleme adımı uygulamamız gerekir:

1. Eğitim veri kümesini ayrı eğitim ve doğrulama bölümlerine ayırın.
2. Eğitim veri kümesindeki benzersiz kelimeleri belirleme
3. Her benzersiz kelimeyi benzersiz bir tamsayı ile eşleyin ve inceleme metnini kodlanmış tamsayılar halinde kodlayın (her benzersiz kelimenin bir dizini)
4. Veri kümesini modele girdi olarak mini partilere bölme

İlk adımla devam edelim: train_dataset'ten bir eğitim ve doğrulama bölümü oluşturmak daha önce okumuştuk:

```
>> ## Adım 1: Veri kümelerini oluşturun
>> from torch.utils.data.dataset import random_split
>>> torch.manual_seed(1)
>> train_dataset, valid_dataset = random_split(
...         list(train_dataset), [20000, 5000])
```

Orijinal eğitim veri kümesi 25.000 örnek içerir. 20.000 örnek eğitim için, 5.000 örnek ise doğrulama için rastgele seçilmiştir.

Verileri bir NN'ye girdi olarak hazırlamak için, *2.* ve *3. adımlarda* belirtildiği gibi sayısal değerlere kodlamamız gerekir. Bunu yapmak için, öncelikle eğitim veri kümesindeki benzersiz kelimeleri (belirteçleri) bulacağız. Benzersiz belirteçleri bulmak Python veri kümelerini kullanabileceğimiz bir işlem olsa da, Python'un standart kütüphanesinin bir parçası olan collections paketindeki Counter sınıfını kullanmak daha verimli olabilir.

Aşağıdaki kodda, benzersiz kelime frekanslarını toplayacak yeni bir Counter nesnesi (token_counts) oluşturacağız. Bu özel uygulamada (ve kelime torbası modelinin aksine), yalnızca benzersiz kelimeler kümesiyle ilgilendiğimizi ve bir yan ürün olarak oluşturulan kelime sayılarına ihtiyaç duymayacağımızı unutmayın. Metni kelimelere (veya belirteçlere) ayırmak için, *Bölüm 8*'de geliştirdiğimiz ve HTML işaretlemelerinin yanı sıra noktalama işaretlerini ve diğer harf olmayan karakterleri de kaldıran belirteçleştirici işlevini yeniden kullanacağız:

Benzersiz belirteçleri toplamak için kod aşağıdaki gibidir:

```
>> ## Adım 2: benzersiz belirteçleri (kelimeleri) bulun
>> import re
>> from collections import Counter, OrderedDict
>>>
>> def tokenizer(metin):
...         metin = re.sub('<[^>]*>', '', metin)
...         emoticons = re.findall(
...             '(?::|;|=)(?:-)?(?:\)|\(|D|P)', text.lower()
...         )
...         metin = re.sub('[\W]+', ' ', metin.lower()) +\
...             ' '.join(emoticons).replace('-', '')
...         tokenized = text.split()
```

```
...         return tokenized
>>>
>> token_counts = Counter()
>> for label, line in train_dataset:
...         tokens = tokenizer(line)
...         token_counts.update(tokens)
>> print('Vocab-size:', len(token_counts))
Vocab-size: 69023
```

Counter hakkında daha fazla bilgi edinmek istiyorsanız, https://docs.python.org/3/library/collections.html#collections.Counter adresindeki belgelere bakın.

Daha sonra, her bir benzersiz kelimeyi benzersiz bir tamsayı ile eşleyeceğiz. Bu, anahtarların benzersiz belirteçler (kelimeler) olduğu ve her anahtarla ilişkili değerin benzersiz bir tamsayı olduğu bir Python sözlüğü kullanılarak manuel olarak yapılabilir. Bununla birlikte, torchtext paketi zaten böyle bir eşleme oluşturmak ve tüm veri kümesini kodlamak için kullanabileceğimiz bir sınıf olan Vocab'ı sağlar. İlk olarak, sıralı sözlük eşleme belirteçlerini karşılık gelen oluşum frekanslarına geçirerek bir vocab nesnesi oluşturacağız (sıralı sözlük, sıralanmış token_counts'tır). İkinci olarak, kelime dağarcığına iki özel belirteç ekleyeceğiz - dolgu ve

```
>> ## Adım 3: Her bir benzersiz belirtecin tam sayılara kodlanması
>> from torchtext.vocab import vocab
>> sorted_by_freq_tuples = sorted(
...         token_counts.items(), key=lambda x: x[1], reverse=True
... )
>> ordered_dict = OrderedDict(sorted_by_freq_tuples)
>> vocab = vocab(ordered_dict)
>> vocab.insert_token("<pad>", 0)
>> vocab.insert_token("<unk>", 1)
>> vocab.set_default_index(1)
```

bilinmeyen belirteç:

vocab nesnesinin nasıl kullanılacağını göstermek için, örnek bir girdi metnini tamsayı değerlerinden oluşan bir listeye dönüştüreceğiz:

```
>> print([vocab[token] for token in ['this', 'is',
...         'bir', 'örnek']])
[11, 7, 35, 457]
```

Doğrulama veya test verilerinde eğitim verilerinde bulunmayan ve dolayısıyla eşlemeye dahil edilmeyen bazı belirteçler olabileceğini unutmayın. Eğer *q* token'ımız varsa (yani, Vocab'a aktarılan token_counts'ın boyutu, ki bu durumda 69.023'tür), daha önce görülmemiş ve dolayısıyla token_counts'a dahil edilmemiş tüm token'lara 1 tamsayısı atanacaktır (bilinmeyen to- ken için bir yer tutucu). Başka bir deyişle, 1 indeksi bilinmeyen sözcükler için ayrılmıştır. Ayrılmış bir diğer değer de 0 tamsayısıdır; bu değer, dizi uzunluğunu ayarlamak için *dolgu belirteci* olarak adlandırılan bir yer tutucu görevi görür. Daha sonra, PyTorch'ta bir RNN modeli oluştururken, bu yer tutucu olan 0'ı daha ayrıntılı olarak ele alacağız.

alinpikbarbell@gmail.com

Veri kümesindeki her metni uygun şekilde dönüştürmek için text_pipeline işlevini tanımlayabiliriz ve label_pipeline işlevi her etiketi 1 veya 0'a dönüştürür:

```
>> ## Adım 3-A: Dönüşüm için fonksiyonları tanımlayın
>> text_pipeline =\
...         lambda x: [vocab[token] for token in tokenizer(x)]
>> label_pipeline = lambda x: 1. if x == 'pos' else 0.
```

DataLoader kullanarak örnek yığınları oluşturacağız ve daha önce bildirilen veri işleme ardışık düzenlerini collate_fn bağımsız değişkenine aktaracağız. Metin kodlama ve etiket dönüştürme işlevini collate_batch işlevine saracağız:

```
>> ## Adım 3-B: kodlama ve dönüştürme işlevini sarın
... def collate_batch(batch):
...         label_list, text_list, lengths = [], [], []
...         for _label, _text in batch:
...             label_list.append(label_pipeline(_label))
...             processed_text = torch.tensor(text_pipeline(_text),
...                                           dtype=torch.int64)
...             text_list.append(processed_text)
...             lengths.append(processed_text.size(0))
...         etiket_listesi = torch.tensor(etiket_listesi)
...         lengths = torch.tensor(lengths)
...         padded_text_list = nn.utils.rnn.pad_sequence(
...             text_list, batch_first=True)
...         return padded_text_list, label_list, lengths
>>>
>> ## Küçük bir parti alın
>> from torch.utils.data import DataLoader
>> dataloader = DataLoader(train_dataset, batch_size=4,
...                         shuffle=False, collate_fn=collate_batch)
```

Şimdiye kadar, kelime dizilerini tamsayı dizilerine ve pos veya neg etiketlerini 1 veya 0'a dönüştürdük. Ancak, çözmemiz gereken bir sorun var - diziler şu anda farklı uzunluklara sahip (aşağıdaki kodu dört örnek için çalıştırmanın sonucunda gösterildiği gibi). Genel olarak, RNN'ler farklı uzunluklardaki dizileri işleyebilse de, bir tensörde verimli bir şekilde saklamak için bir mini partideki tüm dizilerin aynı uzunluğa sahip olduğundan emin olmamız gerekir.

PyTorch, bir yığın içinde birleştirilecek ardışık öğeleri otomatik olarak yer tutucu değerlerle (0'lar) dolduracak ve böylece bir yığın içindeki tüm dizilerin aynı şekle sahip olmasını sağlayacak etkili bir yöntem olan pad_sequence() sağlar. Önceki kodda, eğitim veri kümesinden küçük bir yığın boyutunda bir veri yükleyici oluşturduk ve kendisi de bir pad_sequence() çağrısı içeren collate_batch işlevini uyguladık.

Bununla birlikte, dolgunun nasıl çalıştığını göstermek için, ilk partiyi alacağız ve bunları mini partiler halinde birleştirmeden önce bağımsız elemanların boyutlarını ve sonuçta ortaya çıkan mini partilerin boyutlarını yazdıracağız:

```
>> text_batch, label_batch, length_batch = next(iter(dataloader))
>>  print(text_batch)
tensör([      35, 1742,          7,     449,     723,        6,     302,        4,
...
0,        0,        0,        0,        0,        0,        0,        0]],
>> print(label_batch) tensor([1.,
1., 1., 0.])
>> print(length_batch) tensor([165,
86, 218, 145])
>> print(text_batch.shape)
torch.Size([4, 218])
```

Basılı tensör şekillerinden de görebileceğiniz gibi, ilk partideki sütun sayısı 218'dir; bu da ilk dört örneğin tek bir partide birleştirilmesinden ve bu örneklerin maksimum boyutunun kullanılmasından kaynaklanmaktadır. Bu, bu partideki diğer üç örneğin (uzunlukları sırasıyla 165, 86 ve 145 olan) bu boyuta uyması için gerektiği kadar doldurulduğu anlamına gelir.

Son olarak, her üç veri kümesini de 32 toplu iş boyutuna sahip veri yükleyicilerine bölelim:

```
>>> batch_size = 32
>>  train_dl = DataLoader(train_dataset, batch_size=batch_size,
...                                  shuffle=True, collate_fn=collate_batch)
>> valid_dl = DataLoader(valid_dataset, batch_size=batch_size,
...                                  shuffle=False, collate_fn=collate_batch)
>> test_dl = DataLoader(test_dataset, batch_size=batch_size,
...                                 shuffle=False, collate_fn=collate_batch)
```

Şimdi, veriler aşağıdaki alt bölümlerde uygulayacağımız bir RNN modeli için uygun bir formattadır. Ancak bir sonraki alt bölümde, ilk olarak kelime vektörlerinin boyutluluğunu azaltmak için kullanılan isteğe bağlı ancak şiddetle tavsiye edilen bir ön işleme adımı olan özellik **gömme** işlemini tartışacağız.

Cümle kodlaması için gömme katmanları

Bir önceki adımdaki veri hazırlığı sırasında, aynı uzunlukta diziler oluşturduk. Bu dizilerin elemanları, benzersiz kelimelerin *indekslerine* karşılık gelen tam sayılardı. Bu kelime indisleri birkaç farklı yolla girdi özelliklerine dönüştürülebilir. Naif yollardan biri, indisleri sıfır ve birlerden oluşan vektörlere dönüştürmek için tek vuruşlu kodlama uygulamaktır. Ardından, her kelime, boyutu tüm veri kümesindeki benzersiz kelimelerin sayısı olan bir vektöre eşlenecektir. Benzersiz kelimelerin sayısının (kelime dağarcığının boyutu) 10^4 - 10^5 mertebesinde olabileceği göz önüne alındığında, bu aynı zamanda girdi özelliklerimizin sayısı olacaktır, bu tür özellikler üzerinde eğitilen bir model **boyutsallık lanetinden muzdarip olabilir**. Ayrıca, bu özellikler çok seyrektir çünkü biri hariç hepsi sıfırdır.

olimpikbarbell@gmail.com

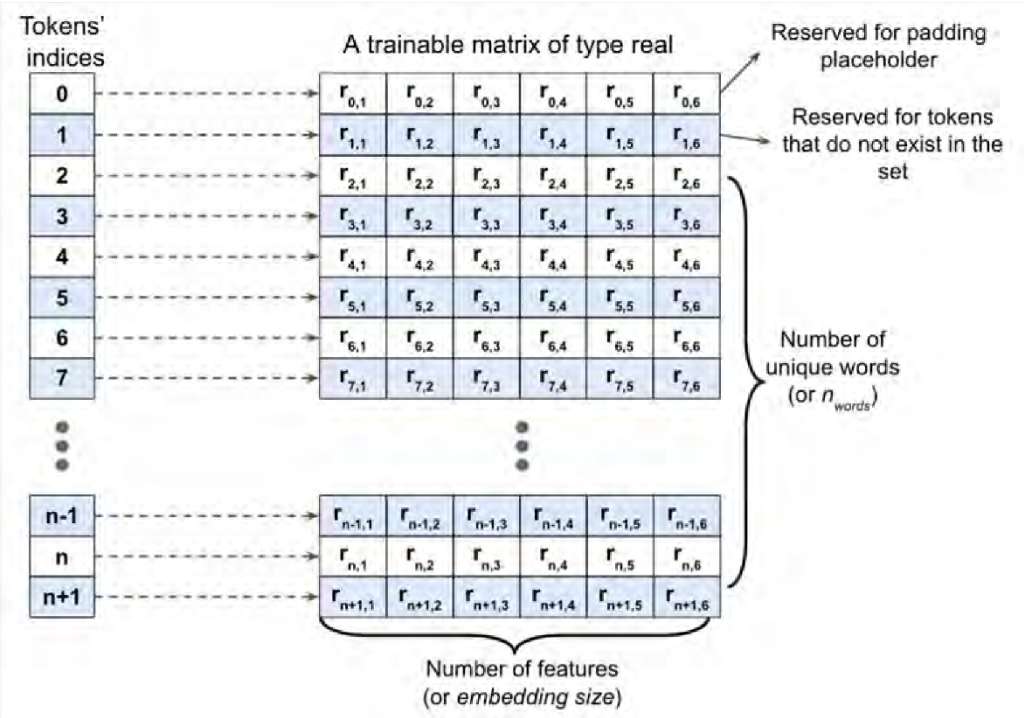
Daha zarif bir yaklaşım, her kelimeyi gerçek değerli elemanları (tam sayı olması gerekmez) olan sabit boyutlu bir vektörle eşlemektir. Bir-sıcak kodlu vektörlerin aksine, sonsuz sayıda gerçek sayıyı temsil etmek için sonlu boyutlu vektörler kullanabiliriz. (Teorik olarak, belirli bir aralıktan, örneğin [-1, 1]'den sonsuz sayıda gerçek sayı çıkarabiliriz).

Veri setimizdeki kelimeleri temsil edecek belirgin özellikleri otomatik olarak öğrenmek için burada kullanabileceğimiz bir özellik öğrenme tekniği olan gömmenin arkasındaki fikir budur. Benzersiz kelime sayısı n_{words} göz önüne alındığında, tüm kelime dağarcığını girdi özellikleri olarak temsil etmek için gömme vektörlerinin boyutunu (diğer bir deyişle gömme boyutunu) benzersiz kelime sayısından ($embedding_dim << n_{words}$) çok daha küçük olacak şekilde seçebiliriz.

Gömmenin tek seferlik kodlamaya göre avantajları aşağıdaki gibidir:

- Boyutluluk lanetinin etkisini azaltmak için özellik uzayının boyutluluğunda bir azalma
- Bir NN'deki gömme katmanı optimize edilebildiğinden (veya öğrenilebildiğinden) göze çarpan özelliklerin çıkarılması

Aşağıdaki şematik gösterim, token indekslerini eğitilebilir bir gömme matrisine eşleyerek gömmenin nasıl çalıştığını göstermektedir:



Şekil 15.10: Yerleştirmenin nasıl çalıştığına dair bir döküm

$N + 2$ boyutunda bir belirteç kümesi verildiğinde (n belirteç kümesinin boyutudur, artı 0 indeksi pad- ding yer tutucusu için ayrılmıştır ve 1 belirteç kümesinde bulunmayan kelimeler içindir), $(n + 2) \times$ *embedding_dim* boyutunda bir gömme matrisi oluşturulacaktır; bu matrisin her satırı bir belirteçle ilişkili sayısal özellikleri temsil etmektedir. Bu nedenle, gömme işlemine girdi olarak bir tamsayı indeksi (i) verildiğinde, matrisin i indeksindeki ilgili satırına bakacak ve sayısal özellikleri döndürecektir. Gömme matrisi, NN modellerimiz için giriş katmanı görevi görür. Pratikte, bir gömme katmanı oluşturmak basitçe nn.Embedding kullanılarak yapılabilir. Bir gömme katmanı oluşturacağımız ve bunu aşağıdaki gibi iki örnekten oluşan bir gruba uygulayacağımız bir örnek görelim:

```
>>> embedding = nn.Embedding(
...         num_embeddings=10,
...         embedding_dim=3,
...         padding_idx=0)
>> # her biri 4 endeksli 2 örnekten oluşan bir grup
>>  text_encoded_input = torch.LongTensor([[1,2,4,5],[4,3,2,0]])
>>  print(embedding(text_encoded_input))

  tensör([[-0.7027,    0.3684, -0.5512],
           [-0.4147,    1.7891, -1.0674],
          [ 1.1400,    0.1595, -1.0167],
          [ 0.0573,   -1.7568, 1 .9067]],
         [[ 1.1400, 0.1595, -1.0167],
          [-0.8165, -0.0946, -0.1881],
          [-0.4147, 1.7891, -1.0674],
          [ 0.0000, 0 . 0 0 0 0 , 0 . 0 0 0 0 ]],
          grad_fn=<EmbeddingBackward>)
```

Bu modelin (gömme katmanı) girdisi *batchsize* × *in- put_length* boyutunda rank 2'ye sahip olmalıdır, burada *input_length* dizilerin uzunluğudur (burada 4). Örneğin, mini partideki bir girdi dizisi <1, 5, 9, 2> olabilir; burada bu dizinin her bir öğesi benzersiz kelimelerin indeksidir. Çıktı, *batchsize* × *input_length* × *embedding_dim* boyutuna sahip olacaktır; burada *em- bedding_dim, gömme* özelliklerinin boyutudur (burada 3 olarak ayarlanmıştır). Gömme katmanına sağlanan diğer argüman olan num_embeddings, modelin girdi olarak alacağı benzersiz tamsayı değerlerine karşılık gelir (örneğin, $n + 2$, burada 10 olarak ayarlanmıştır). Dolayısıyla, bu durumda gömme matrisi 10×3 boyutundadır.

padding_idx, belirtilirse eğitim sırasında gradyan güncellemelerine katkıda bulunmayacak olan dolgu için belirteç indeksini (burada 0) gösterir. Örneğimizde, ikinci örneğin orijinal dizisinin uzunluğu 3'tür ve bunu 1 tane daha 0 elemanı ile doldurduk. Dolgulu elemanın gömme çıktısı [0, 0, 0]'dır.

olimpikbarbell@gmail.com

Bir RNN modeli oluşturma

Şimdi bir RNN modeli oluşturmaya hazırız. nn.Module sınıfını kullanarak gömme katmanını, RNN'nin tekrarlayan katmanlarını ve tam bağlı tekrarlamayan katmanları birleştirebiliriz. Tekrarlayan katmanlar için aşağıdaki uygulamalardan herhangi birini kullanabiliriz:

- RNN: normal bir RNN katmanı, yani tam bağlantılı bir tekrarlayan katman
- LSTM: uzun vadeli bağımlılıkları yakalamak için yararlı olan uzun kısa vadeli bir bellek RNN
- GRU: *Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation*'da (*K. Cho ve diğerleri,* 2014) (https:// arxiv.org/abs/1406.1078v3) LSTM'lere alternatif olarak önerildiği gibi, geçitli tekrarlayan birime sahip tekrarlayan bir katman

Bu tekrarlayan katmanlardan birini kullanarak çok katmanlı bir RNN modelinin nasıl oluşturulabileceğini görmek için aşağıdaki örnekte, RNN türünde iki tekrarlayan katmana sahip bir RNN modeli oluşturacağız. Son olarak, tahmin olarak tek bir çıktı değeri döndürecek olan çıktı katmanı olarak tekrarlamayan tam bağlı bir katman ekleyeceğiz:

```
>> sınıf RNN(nn.Module):
...     def init (self, input_size, hidden_size):
...         super(). init ()
...         self.rnn = nn.RNN(input_size, hidden_size, num_layers=2,
...                                 batch_first=True)
...         # self.rnn = nn.GRU(input_size, hidden_size, num_layers,
...         #                         batch_first=True)
...         # self.rnn = nn.LSTM(input_size, hidden_size, num_layers,
...         #                         batch_first=True)
...         self.fc = nn.Linear(hidden_size, 1)
...
...     def ileri(self, x):
...         _, hidden = self.rnn(x)
...         out = hidden[-1, :, :] # son gizli durumu kullanırız
...                                 # olarak son gizli katmandan
...                                 # tam bağlı # katmanın girdisi
...         out = self.fc(out)
...         geri dön
...
>>>
>>> model = RNN(64, 32)
>> print(model)
>>> model(torch.randn(5, 3, 64))
RNN(
  (rnn): RNN(64, 32, num_layers=2, batch_first=True) (fc):
  Linear(in_features=32, out_features=1, bias=True)
)
```

```
tensör([[ 0.0010],
        [ 0.2478],
        [ 0.0573],
        [ 0.1637],
          [-0.0073]], grad_fn=<AddmmBackward>)
```

Gördüğünüz gibi, bu tekrarlayan katmanları kullanarak bir RNN modeli oluşturmak oldukça basittir. Bir sonraki alt bölümde, duygu analizi görevimize geri döneceğiz ve bunu çözmek için bir RNN modeli oluşturacağız.

Duygu analizi görevi için bir RNN modeli oluşturma

Çok uzun dizilerimiz olduğundan, uzun menzilli etkileri hesaba katmak için bir LSTM katmanı kullanacağız. Duygu analizi için, 20 özellik boyutunda (embed_dim=20) kelime katıştırmaları üreten bir katıştırma katmanı ile başlayan bir RNN modeli oluşturacağız. Ardından, LSTM tipi bir tekrarlayan katman eklenecektir. Son olarak, gizli katman olarak tam bağlı bir katman ve çıktı katmanı olarak lojistik sigmoid aktivasyonu yoluyla tahmin olarak tek bir sınıf-üyelik olasılık değeri döndürecek olan başka bir tam bağlı katman ekleyeceğiz:

```
>> sınıf RNN(nn.Module):
...     def init (self, vocab_size, embed_dim, rnn_hidden_size,
...                 fc_hidden_size):
...         super(). init ()
...         self.embedding = nn.Embedding(vocab_size,
...                                       embed_dim,
...                                       padding_idx=0)
...         self.rnn = nn.LSTM(embed_dim, rnn_hidden_size,
...                            batch_first=True)
...         self.fc1 = nn.Linear(rnn_hidden_size, fc_hidden_size)
...         self.relu = nn.ReLU()
...         self.fc2 = nn.Linear(fc_hidden_size, 1)
...         self.sigmoid = nn.Sigmoid()
...
...     def forward(self, text, lengths):
...         out = self.embedding(text)
...         out = nn.utils.rnn.pack_padded_sequence(
...             out, lengths.cpu().numpy(), enforce_sorted=False,
...             batch_first=True
...         )
...         out, (hidden, cell) = self.rnn(out)
...         out = hidden[-1, :, :]
...         out = self.fc1(out) out
...         = s e l f . relu(out) out
...         = s e l f . fc2(out)
...         out = self.sigmoid(out)
...         geri dön
```

 olimpikbarbell@gmail.com

```
>>>
>> vocab_size = len(vocab)
>>> embed_dim = 20
>> rnn_hidden_size = 64
>> fc_hidden_size = 64
>>> torch.manual_seed(1)
>>> model = RNN(vocab_size, embed_dim,
                rnn_hidden_size, fc_hidden_size)
>> model RNN(
  (gömme): Embedding(69025, 20, padding_idx=0) (rnn):
  LSTM(20, 64, batch_first=True)
  (fc1): Linear(in_features=64, out_features=64, bias=True)
  (relu): ReLU()
  (fc2): Linear(in_features=64, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

Şimdi modeli bir epok için verilen veri kümesi üzerinde eğitmek ve sınıflandırma doğruluğunu ve kaybını döndürmek için train işlevini geliştireceğiz:

```
>> def train(dataloader):
...     model.train()
...     total_acc, total_loss = 0, 0
...     for text_batch, label_batch, lengths in dataloader:
...         optimizer.zero_grad()
...         pred = model(text_batch, lengths)[:, 0] loss
...         = loss_fn(pred, label_batch)
...         loss.backward()
...         optimizer.step()
...         total_acc += (
...             (pred >= 0.5).float() == label_batch
...         ).float().sum().item()
...         total_loss += loss.item()*label_batch.size(0)
...     return total_acc/len(dataloader.dataset), \
...             total_loss/len(dataloader.dataset)
```

Benzer şekilde, modelin belirli bir veri kümesi üzerindeki performansını ölçmek için değerlendirme işlevini geliştireceğiz:

```
>> def evaluate(dataloader):
...     model.eval()
...     total_acc, total_loss = 0, 0
```

```
...     torch.no_grad() ile:
...         for text_batch, label_batch, lengths in dataloader:
...             pred = model(text_batch, lengths)[:, 0]
...             loss = loss_fn(pred, label_batch) total_acc
...             += (
...                 (pred>=0.5).float() == label_batch
...             ).float().sum().item()
...             total_loss += loss.item()*label_batch.size(0)
...     return total_acc/len(dataloader.dataset), \
...            total_loss/len(dataloader.dataset)
```

Bir sonraki adım bir kayıp fonksiyonu ve optimizer (Adam optimizer) oluşturmaktır. Tek bir sınıf-üyelik olasılığı çıktısına sahip ikili bir sınıflandırma için, kayıp fonksiyonu olarak ikili çapraz entropi kaybını (BCELoss) kullanıyoruz:

```
>> kayıp_fn = nn.BCELoss()
>>> optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Şimdi modeli 10 epok için eğiteceğiz ve eğitim ve doğrulama performanslarını göstereceğiz:

```
>> num_epochs = 10
>>> torch.manual_seed(1)
>> for epoch in range(num_epochs):
...     acc_train, loss_train = train(train_dl)
...     acc_valid, loss_valid = evaluate(valid_dl)
...     print(f'Epoch {epoch} accuracy: {acc_train:.4f}'
...           f' val_accuracy: {acc_valid:.4f}')
Epoch 0 doğruluk: 0.5843 val_accuracy: 0.6240
Epoch 1 doğruluk: 0.6364 val_accuracy: 0.6870
Epoch 2 doğruluk: 0.8020 val_accuracy: 0.8194
Epoch 3 doğruluk: 0.8730 val_accuracy: 0.8454
Epoch 4 doğruluk: 0.9092 val_accuracy: 0.8598
Epoch 5 doğruluk: 0.9347 val_accuracy: 0.8630
Epoch 6 doğruluk: 0.9507 val_accuracy: 0.8636
Epoch 7 doğruluk: 0.9655 val_accuracy: 0.8654
Epoch 8 doğruluk: 0.9765 val_accuracy: 0.8528
Epoch 9 doğruluk: 0.9839 val_accuracy: 0.8596
```

Bu modeli 10 epok boyunca eğittikten sonra, test verileri üzerinde değerlendireceğiz:

```
>> acc_test, _ = evaluate(test_dl)
>> print(f'test_accuracy: {acc_test:.4f}')
test_accuracy: 0.8512
```

 olimpikbarbell@gmail.com

Yüzde 85 doğruluk göstermiştir. (Bu sonucun IMDb veri kümesinde kullanılan son teknoloji yöntemlerle karşılaştırıldığında en iyisi olmadığını unutmayın. Amaç sadece bir RNN'nin PyTorch'ta nasıl çalıştığını göstermekti).

Çift yönlü RNN hakkında daha fazla bilgi

Buna ek olarak, LSTM'nin çift yönlü yapılandırmasını True olarak ayarlayacağız, bu da tekrarlayan katmanın giriş dizilerini her iki yönden, baştan sona ve ters yönde geçmesini sağlayacaktır:

```
>>> sınıf RNN(nn.Module):
...     def init (self, vocab_size, embed_dim,
...                 rnn_hidden_size, fc_hidden_size):
...         super(). init ()
...         self.embedding = nn.Embedding( vocab_size,
...             embed_dim, padding_idx=0
...         )
...         self.rnn = nn.LSTM(embed_dim, rnn_hidden_size,
...                         batch_first=True, bidirectional=True)
...         self.fc1 = nn.Linear(rnn_hidden_size*2, fc_hidden_size)
...         self.relu = n n . ReLU()
...         self.fc2 = nn.Linear(fc_hidden_size, 1)
...         self.sigmoid = nn.Sigmoid()
...
...     def forward(self, text, lengths): out
...         = self.embedding(text)
...         out = nn.utils.rnn.pack_padded_sequence(
...             out, lengths.cpu().numpy(), enforce_sorted=False,
...             batch_first=True
...         )
...         _, (gizli, hücre) = self.rnn(out)
...         out = torch.cat((gizli[-2, :, :],
...                         hidden[-1, :, :]), dim=1)
...         out = self.fc1(out)
...         out = self.relu(out)
...         out = s e l f . fc2(out)
...         out = self.sigmoid(out)
>>>         geri dön
>>> torch.manual_seed(1)
>>> model = RNN(vocab_size, embed_dim,
...             rnn_hidden_size, fc_hidden_size)
>>> model
```

```
RNN(
  (gömme): Gömme(69025, 20, padding_idx=0)
  (rnn): LSTM(20, 64, batch_first=True, bidirectional=True)
  (fc1): Linear(in_features=128, out_features=64, bias=True)
  (relu): ReLU()
  (fc2): Linear(in_features=64, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

Çift yönlü RNN katmanı her bir girdi dizisi üzerinde iki geçiş yapar: bir ileri geçiş ve bir geri veya ters geçiş (bunun geri yayılma bağlamındaki ileri ve geri geçişlerle karıştırılmaması gerektiğini unutmayın). Bu ileri ve geri geçişlerden elde edilen gizli durumlar genellikle tek bir gizli durum halinde birleştirilir. Diğer birleştirme modları arasında toplama, çarpma (iki geçişin sonuçlarını çarpma) ve ortalama alma (ikisinin ortalamasını alma) yer alır.

Normal RNN gibi diğer tekrarlayan katman türlerini de deneyebiliriz. Ancak, ortaya çıktığı üzere, normal tekrarlayan katmanlarla oluşturulan bir model iyi bir tahmin performansına ulaşamayacaktır (eğitim verilerinde bile). Örneğin, önceki koddaki çift yönlü LSTM katmanını tek yönlü bir nn.RNN (nn.LSTM yerine) katmanıyla değiştirmeyi denerseniz ve modeli tam uzunluktaki diziler üzerinde eğitirseniz, eğitim sırasında kaybın azalmayacağını bile gözlemleyebilirsiniz. Bunun nedeni, bu veri kümesindeki dizilerin çok uzun olmasıdır, bu nedenle RNN katmanına sahip bir model uzun vadeli bağımlılıkları öğrenemez ve kaybolan veya patlayan gradyan sorunlarından muzdarip olabilir.

Proje iki - PyTorch'ta karakter düzeyinde dil modelleme

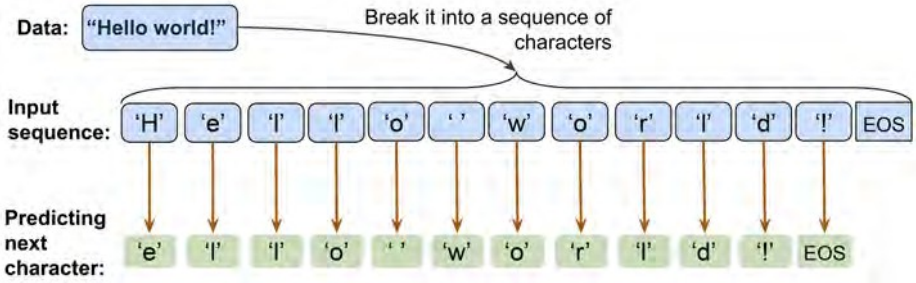
Dil modelleme, makinelerin İngilizce cümleler üretmek gibi insan diliyle ilişkili görevleri yerine getirmesini sağlayan büyüleyici bir uygulamadır. Bu alandaki ilginç çalışmalardan biri *Ilya Sutskever*, *James Martens* ve *Geoffrey E. Hinton* tarafından hazırlanan *Gen- erating Text with Recurrent Neural Networks, Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 2011 (https://pdfs. semanticscholar.org/93c2/0e38c85b69fc2d2eb314b3c1217913f7db11.pdf).

Şimdi oluşturacağımız modelde girdi bir metin belgesidir ve amacımız girdi belgesine benzer tarzda yeni metin üretebilen bir model geliştirmektir. Bu tür girdilere örnek olarak bir kitap veya belirli bir programlama dilindeki bir bilgisayar programı verilebilir.

Karakter düzeyinde dil modellemesinde girdi, her seferinde bir karakter olmak üzere ağımıza beslenen bir dizi karaktere ayrılır. Ağ, bir sonraki karakteri tahmin etmek için her yeni karakteri daha önce görülen karakterlerin hafızası ile birlikte işleyecektir.

ilinpikbarbell@gmail.com

Şekil 15.11'de karakter düzeyinde bir dil modelleme örneği gösterilmektedir (EOS'nin "dizinin sonu" anlamına geldiğine dikkat edin):



Şekil 15.11: Karakter düzeyinde dil modellemesi

Bu uygulamayı üç ayrı adıma ayırabiliriz: verilerin hazırlanması, RNN modelinin oluşturulması ve yeni metin oluşturmak için sonraki karakter tahmini ve örnekleme yapılması.

Veri kümesini ön işleme

Bu bölümde, verileri karakter düzeyinde dil modellemesi için hazırlayacağız.

Girdi verilerini elde etmek için, binlerce ücretsiz e-kitap sağlayan https://www.gutenberg.org/ adresindeki Project Gutenberg web sitesini ziyaret edin. Örneğimiz için, Jules Verne tarafından yazılan *Gizemli Ada* (1874 yılında yayınlanmıştır) kitabını düz metin biçiminde https://www.gutenberg.org/ files/1268/1268-0.txt adresinden indirebilirsiniz.

Bu bağlantının sizi doğrudan indirme sayfasına götüreceğini unutmayın. MacOS veya Linux işletim sistemi kullanıyorsanız, dosyayı terminalde aşağıdaki komutla indirebilirsiniz:

```
curl -O https://www.gutenberg.org/files/1268/1268-0.txt
```

Bu kaynağın gelecekte kullanılamaz hale gelmesi durumunda, bu metnin bir kopyası da https://github.com/rasbt/machine-learning-book adresindeki kitabın kod deposunda bu bölümün kod dizininde yer almaktadır.

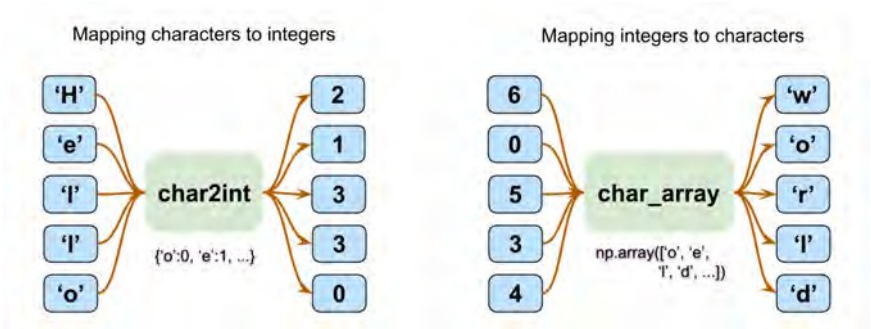
Veri kümesini indirdikten sonra, onu bir Python oturumunda düz metin olarak okuyabiliriz. Aşağıdaki kodu kullanarak, metni doğrudan indirilen dosyadan okuyacağız ve başından ve sonundan bazı kısımları çıkaracağız (bunlar Gutenberg projesinin belirli açıklamalarını içerir). Ardından, bu metinde gözlemlenen *benzersiz* karakter kümesini temsil eden char_set adlı bir

```
>> import numpy as np
>> ## Metin okuma ve işleme
>> with open('1268-0.txt', 'r', encoding="utf8") as fp:
...         text=fp.read()
>> start_indx = text.find('GİZEMLİ ADA')
>> end_indx = text.find('Gutenberg Projesi'nin Sonu')
>>> metin = metin[başlangıç_indx:bitiş_indx]
>> char_set = set(text)
```

Python değişkeni oluşturacağız:

```
>> print('Toplam Uzunluk:',
len(metin)) Toplam Uzunluk:
1112350
>> print('Benzersiz Karakterler:', len(char_set))
Benzersiz Karakterler: 80
```

Metni indirdikten ve ön işlemden geçirdikten sonra, toplamda 1.112.350 karakterden ve 80 benzersiz karakterden oluşan bir diziye sahibiz. Bununla birlikte, çoğu NN kütüphanesi ve RNN uygulaması dize biçimindeki giriş verileriyle başa çıkamaz, bu yüzden metni sayısal bir biçime dönüştürmemiz gerekir. Bunu yapmak için, her karakteri bir tamsayıya, char2int'e eşleyen basit bir Python sözlüğü oluşturacağız. Ayrıca modelimizin sonuçlarını tekrar metne dönüştürmek için bir ters eşlemeye ihtiyacımız olacak. Tersi, tamsayı anahtarlarını karakter değerleriyle ilişkilendiren bir sözlük kullanılarak yapılabilse de, bir NumPy dizisi kullanmak ve diziyi indeksleri bu benzersiz karakterlerle eşlemek için indekslemek daha verimlidir. *Şekil 15.12'de* "Hello" ve "world" kelimeleri için karakterleri tamsayılara dönüştürme ve tersini yapma örneği gösterilmektedir:



Şekil 15.12: Karakter ve tamsayı eşlemeleri

Karakterleri tamsayılara eşlemek için sözlük oluşturmak ve önceki şekilde gösterildiği gibi bir NumPy dizisini indeksleyerek ters eşleme yapmak aşağıdaki gibidir:

```
>> chars_sorted = sorted(char_set)
>> char2int = {ch:i for i,ch in enumerate(chars_sorted)}
>> char_array = np.array(chars_sorted)
>> text_encoded = np.array(
...        [char2int[ch] for ch in text],
...        dtype=np.int32
... )
>> print('Metin kodlanmış şekil:', text_encoded.shape)
Metin kodlanmış şekil: (1112350,)
>> print(text[:15], '== Kodlama ==>', text_encoded[:15])
>> print(text_encoded[15:21], '== Ters ==>',
...            ''.join(char_array[text_encoded[15:21]]))
THE MYSTERIOUS == Kodlama ==> [44 32 29 1 37 48 43 44 29 42 33 39 45 43 1]
[33 43 36 25 38 28] == Ters ==> ADA
```

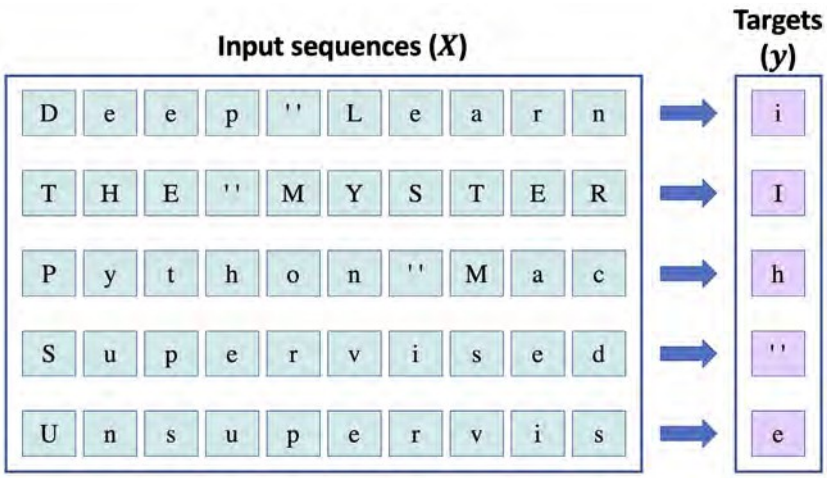
 alinpikbarbell@gmail.com

text_encoded NumPy dizisi, metindeki tüm karakterler için kodlanmış değerleri içerir. Şimdi, bu diziden ilk beş karakterin eşlemelerini yazdıracağız:

```
>> for ex in text_encoded[:5]:
...     print('{} -> {}'.format(ex, char_array[ex]))
44 -> T
32 -> H
29 -> E
1 ->
37 -> M
```

Şimdi geri adım atalım ve yapmaya çalıştığımız şeyin büyük resmine bakalım. Metin oluşturma görevi için, problemi bir sınıflandırma görevi olarak formüle edebiliriz.

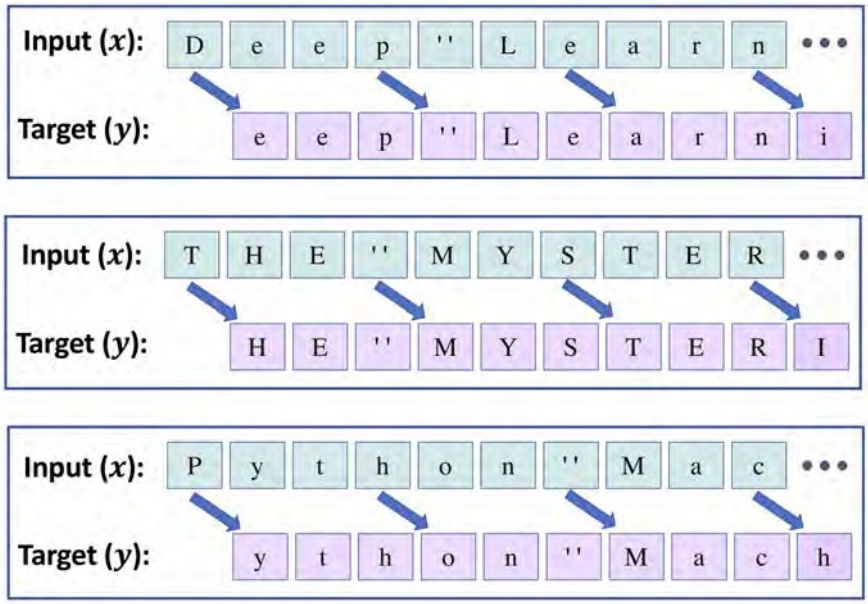
Şekil 15.13'te gösterildiği gibi, tamamlanmamış bir metin karakterleri dizisine sahip olduğumuzu varsayalım:



Şekil 15.13: Bir metin dizisi için bir sonraki karakteri tahmin etme

Şekil 15.13'te, sol taraftaki kutuda gösterilen dizileri girdi olarak kabul edebiliriz. Yeni metin oluşturmak için amacımız, girdi dizisinin tamamlanmamış bir metni temsil ettiği belirli bir girdi dizisinin bir sonraki karakterini tahmin edebilen bir model tasarlamaktır. Örneğin, "Deep Learn "ü gördükten sonra, model bir sonraki karakter olarak "i "yi tahmin etmelidir. 80 benzersiz karakterimiz olduğu göz önüne alındığında, bu problem çok sınıflı bir sınıflandırma görevi haline gelir.

Uzunluğu 1 olan bir diziden (yani tek bir harften) başlayarak, *Şekil 15.14'te* gösterildiği gibi, bu çok sınıflı sınıflandırma yaklaşımına dayalı olarak yinelemeli bir şekilde yeni metin oluşturabiliriz:



Şekil 15.14: Bu çok sınıflı sınıflandırma yaklaşımına dayalı olarak bir sonraki metnin oluşturulması

PyTorch'ta metin oluşturma görevini uygulamak için önce dizi uzunluğunu 40'a kırpalım. Bu, girdi tensörü ***x'***in 40 belirteçten oluştuğu anlamına gelir. Pratikte, dizi uzunluğu üretilen metnin kalitesini etkiler. Daha uzun diziler daha anlamlı cümlelerle sonuçlanabilir. Ancak daha kısa diziler için model, bağlamı çoğunlukla göz ardı ederken tek tek kelimeleri doğru bir şekilde yakalamaya odaklanabilir. Daha uzun diziler genellikle daha anlamlı cümlelerle sonuçlansa da, belirtildiği gibi, uzun diziler için RNN modeli uzun menzilli bağımlılıkları yakalamakta sorun yaşayacaktır. Bu nedenle, pratikte, dizi uzunluğu için tatlı bir nokta ve iyi bir değer bulmak, deneysel olarak değerlendirmemiz gereken bir hiperparametre optimizasyon problemidir. Burada, iyi bir denge sunduğu için 40'ı seçeceğiz.

Önceki şekilde görebileceğiniz gibi, ***x*** girişleri ve ***y*** hedefleri bir karakter kaydırılmıştır. Bu nedenle, metni 41 boyutlu parçalara böleceğiz: ilk 40 karakter girdi dizisini, ***x,*** ve son 40 öğe hedef dizisini, ***y,*** oluşturacaktır.

 olimpikbarbell@gmail.com

Kodlanmış metnin tamamını orijinal sırasına göre text_encoded içinde zaten saklamıştık. İlk olarak her biri 41 karakterden oluşan metin parçaları oluşturacağız. Eğer 41 karakterden daha kısa ise son parçadan da kurtulacağız. Sonuç olarak, text_chunks adlı yeni yığın veri kümesi her zaman 41 boyutunda diziler içerecektir. 41 karakterlik parçalar daha sonra ***x*** dizisini (yani girdi) ve ***y*** dizisini (yani hedef) oluşturmak için kullanılacak ve her ikisi de 40 öğeye sahip olacaktır. Örneğin, ***x dizisi*** [0, 1, ..., 39] indisli elemanlardan oluşacaktır. Ayrıca, ***y*** dizisi ***x'e*** göre bir konum kaydırılacağından, karşılık gelen indisleri [1, 2, ..., 40] olacaktır. Daha sonra, kendi tanımladığımız bir Dataset sınıfını uygulayarak sonucu bir Dataset nesnesine dönüştüreceğiz:

```
>> import torch
>> from torch.utils.data import Dataset
>> seq_length = 40
>> chunk_size = seq_length + 1
>>> text_chunks = [text_encoded[i:i+chunk_size]
...                    for i in range(len(text_encoded)-chunk_size+1)]
>> from torch.utils.data import Dataset
>> class TextDataset(Dataset):
...     def init (self, text_chunks):
...         self.text_chunks = text_chunks
...
...     def len (self):
...         return len(self.text_chunks)
...
...     def getitem (self, idx):
...         text_chunk = self.text_chunks[idx]
...         return text_chunk[:-1].long(), text_chunk[1:].long()
>>>
> >  seq_dataset = TextDataset(torch.tensor(text_chunks))
```

Bu dönüştürülmüş veri kümesinden bazı örnek dizilere bir göz atalım:

```
>> for i, (seq, target) in enumerate(seq_dataset):
...     print(' Girdi (x): ',
...             repr(''.join(char_array[seq]))
...     print('Hedef (y): ',
...             repr(''.join(char_array[target]))
...     print()
...     eğer i == 1:
...         Mola
 Girdi (x): 'THE MYSTERIOUS ISLAND ***\n\n\n\nProduced b'
Hedef (y): 'GİZEMLİ ADA ***\n\n\n\nÜreten'

 Girdi (x): 'HE MYSTERIOUS ISLAND ***\n\n\n\nProduced by'
Hedef (y): 'E GİZEMLİ ADA ***\n\n\n\nÜreten'
```

Son olarak, veri setinin hazırlanmasındaki son adım, bu veri setinin mini partilere dönüştürülmesidir:

```
>> from torch.utils.data import DataLoader
>>> batch_size = 64
>>> torch.manual_seed(1)
>> seq_dl = DataLoader(seq_dataset, batch_size=batch_size,
...                                shuffle=True, drop_last=True)
```

Karakter düzeyinde bir RNN modeli oluşturma

Veri seti hazır olduğuna göre, modeli oluşturmak nispeten daha kolay olacaktır:

```
>> import torch.nn as nn
>> sınıf RNN(nn.Module):
...     def init (self, vocab_size, embed_dim, rnn_hidden_size):
...         super(). init ()
...         self.embedding = nn.Embedding(vocab_size, embed_dim)
...         self.rnn_hidden_size = rnn_hidden_size
...         self.rnn = nn.LSTM(embed_dim, rnn_hidden_size,
...                                batch_first=True)
...         self.fc = nn.Linear(rnn_hidden_size, vocab_size)
...
...     def ileri(self, x, gizli, hücre):
...         out = self.embedding(x).unsqueeze(1)
...         out, (hidden, cell) = self.rnn(out, (hidden, cell))
...         out = self.fc(out).reshape(out.size(0), -1)
...         dışarı dön, gizli, hücre
...
...     def init_hidden(self, batch_size):
...         hidden = torch.zeros(1, batch_size, self.rnn_hidden_size)
...         cell = torch.zeros(1, batch_size, self.rnn_hidden_size)
...         return hidden, cell
```

Yeni metin oluşturmak için model tahminlerinden örnekleme yapabilmemiz için logitlerin modelin çıktıları olması gerektiğine dikkat edin. Bu örnekleme kısmına daha sonra geleceğiz.

Ardından, model parametrelerini belirleyebilir ve bir RNN modeli oluşturabiliriz:

```
>> vocab_size = len(char_array)
>>> embed_dim = 256
>> rnn_hidden_size = 512
>>> torch.manual_seed(1)
>>> model = RNN(vocab_size, embed_dim, rnn_hidden_size)
>> model RNN(
  (gömme): Gömme(80, 256)
```

 olimpikbarbell@gmail.com

```
    (rnn): LSTM(256, 512, batch_first=True)
    (fc): Linear(in_features=512, out_features=80, bias=True)
  )
```

Bir sonraki adım bir kayıp fonksiyonu ve optimizer (Adam optimizer) oluşturmaktır. Her hedef karakter için tek bir logit çıktısı olan çok sınıflı bir sınıflandırma için (vocab_size=80 sınıfımız var), kayıp fonksiyonu olarak CrossEntropyLoss kullanıyoruz:

```
>> kayıp_fn = nn.CrossEntropyLoss()
>>> optimizer = torch.optim.Adam(model.parameters(), lr=0.005)
```

Şimdi modeli 10.000 epoch için eğiteceğiz. Her epokta, veri yükleyiciden rastgele seçilen yalnızca bir yığın kullanacağız, seq_dl. Ayrıca her 500 epok için eğitim kaybını da göstereceğiz:

```
>> num_epochs = 10000
>>> torch.manual_seed(1)
>> for epoch in range(num_epochs):
...         hidden, cell = model.init_hidden(batch_size)
...         seq_batch, target_batch = next(iter(seq_dl))
...         optimizer.zero_grad()
...         kayıp = 0
...         for c in range(seq_length):
...             pred, hidden, cell = model(seq_batch[:, c], hidden, cell)
...             loss += loss_fn(pred, target_batch[:, c])
...         loss.backward()
...         optimizer.step()
...         loss = loss.item()/seq_length
...         if epoch % 500 == 0:
...             print(f'Epoch {epoch} loss: {loss:.4f}')
Epoch 0 loss: 1.9689
Epoch 500 kaybı: 1.4064
Epoch  1000 kayıp 1.3155
                :
Epoch  1500 kayıp 1.2414
                :
Epoch  2000 kayıp 1.1697
                :
Epoch  2500 kayıp 1.1840
                :
Epoch  3000 kayıp 1.1469
                :
Epoch  3500 kayıp 1.1633
                :
Epoch  4000 kayıp 1.1788
                :
Epoch  4500 kayıp 1.0828
                :
Epoch  5000 kayıp 1.1164
                :
```

6587191CA584E


```
Epoch 5500 kayıp 1.0821
:
Epoch 6000 kayıp 1.0764
:
```

 olimpikbarbell@gmail.com

```
Epoch 6500 kaybı: 1.0561
Epoch 7000 kaybı: 1.0631
Epoch 7500 kaybı: 0.9904
Epoch 8000 kaybı: 1.0053
Epoch 8500 kaybı: 1.0290
Epoch 9000 kaybı: 1.0133
Epoch 9500 kaybı: 1.0047
```

Daha sonra, belirli bir kısa dizeden başlayarak yeni metin oluşturmak için modeli değerlendirebiliriz. Bir sonraki bölümde, eğitilmiş modeli değerlendirmek için bir fonksiyon tanımlayacağız.

Değerlendirme aşaması - yeni metin pasajları oluşturma

Önceki bölümde eğittiğimiz RNN modeli, her bir benzersiz karakter için 80 boyutunda logit döndürmektedir. Bu logitler, softmax fonksiyonu aracılığıyla, belirli bir karakterin bir sonraki karakter olarak karşılaşılma olasılığına kolayca dönüştürülebilir. Dizideki bir sonraki karakteri tahmin etmek için, en yüksek olasılığa sahip karakteri seçmeye eşdeğer olan maksimum logit değerine sahip öğeyi seçebiliriz. Ancak, her zaman en yüksek olasılığa sahip karakteri seçmek yerine, çıktılardan (rastgele) *örnekleme* yapmak istiyoruz; aksi takdirde, model her zaman aynı metni üretecektir. PyTorch zaten kategorik bir dağılımdan rastgele örnekler çekmek için kullanabileceğimiz torch.distributions.categorical.Categorical adlı bir sınıf sağlar. Bunun nasıl çalıştığını görmek için, girdi logitleri [1, 1, 1] olan üç kategoriden [0, 1, 2] bazı rastgele örnekler

```
>> from torch.distributions.categorical import Categorical
>>> torch.manual_seed(1)
>>> logits = torch.tensor([[1.0, 1.0, 1.0]])
>> print('Olasılıklar:',
...          nn.functional.softmax(logits, dim=1).numpy()[0])
Olasılıklar: [0.33333334 0.33333334 0.33333334]
>>> m = Kategorik(logits=logits)
>>> örnekler = m.örnek((10,))
>> print(samples.numpy())
[[0]
 [0]
 [0]
 [0]
 [1]
 [0]
 [1]
 [2]
 [1]
 [1]]
```

oluşturalım:

6587191CA584E

Gördüğünüz gibi, verilen logitlerle kategoriler aynı olasılıklara sahiptir (yani, eşit olasılıklı kategoriler). Bu nedenle, büyük bir örneklem boyutu kullanırsak (*num_samples* → ∞), her kategorinin oluşum sayısının örneklem boyutunun ≈ 1/3'üne ulaşmasını bekleriz. Logitleri [1, 1, 3] olarak değiştirirsek, kategori 2 için daha fazla olay gözlemlemeyi bekleriz (bu dağılımdan çok sayıda örnek çekildiğinde):

```
>>> torch.manual_seed(1)
>>> logits = torch.tensor([[1.0, 1.0, 3.0]])
>> print('Olasılıklar:', nn.functional.softmax(logits, dim=1).numpy()[0]) Olasılıklar:
[0.10650698 0.10650698 0.78698605]
>>  m = Kategorik(logits=logits)
>>> örnekler = m.örnek((10,))
>> print(samples.numpy())
[[0]
 [2]
 [2]
 [1]
 [2]
 [1]
 [2]
 [2]
 [2]
 [2]]
```

Categorical kullanarak, modelimiz tarafından hesaplanan logitlere dayalı örnekler oluşturabiliriz.

Kısa bir başlangıç dizesi (starting_str) alan ve başlangıçta giriş dizesine ayarlanan yeni bir dize (generated_str) üreten sample() adlı bir işlev tanımlayacağız. starting_str bir dizi tamsayıya (encoded_input) kodlanır. encoded_input, gizli durumları güncellemek için her seferinde bir karakter olmak üzere RNN modeline aktarılır. Kodlanmış_girdi'nin son karakteri yeni bir karakter oluşturmak için modele aktarılır. RNN modelinin çıktısının, model tarafından girdi dizisi gözlemlendikten sonra bir sonraki karakter için logitleri (burada, olası karakterlerin toplam sayısı olan 80 boyutunda bir vektör) temsil ettiğini unutmayın.

Burada, sadece yeni bir örnek oluşturmak için Categorical sınıfına aktarılan logits çıktısını (yani, $\boldsymbol{o}^{(T)}$) kullanıyoruz. Bu yeni örnek bir karaktere dönüştürülür ve daha sonra oluşturulan dizinin (generated_text) sonuna eklenerek uzunluğu 1 artırılır. Daha sonra, oluşturulan dizinin uzunluğu istenen değere ulaşana kadar bu işlem tekrarlanır. Yeni öğeler üretmek için üretilen diziyi girdi olarak kullanma işlemine **otomatik regresyon** denir.

sample() fonksiyonunun kodu aşağıdaki gibidir:

```
>>  def sample(model, starting_str,
...                 len_generated_text=500,
...                 scale_factor=1.0):
```

 olimpikbarbell@gmail.com

```
...     encoded_input = torch.tensor(
...         [char2int[s] for s in starting_str]
...     )
...     encoded_input = torch.reshape(
...         encoded_input, (1, -1)
...     )
...     generated_str = starting_str
...
...     model.eval()
...     hidden, cell = model.init_hidden(1)
...     for c in range(len(starting_str)-1):
...         _, gizli, hücre = model(
...             encoded_input[:, c].view(1), hidden, cell
...         )
...
...     last_char = encoded_input[:, -1]
...     for i in range(len_generated_text):
...         logits, hidden, cell = model(
...             last_char.view(1), gizli, hücre
...         )
...         logits = torch.squeeze(logits, 0)
...         scaled_logits = logits * scale_factor m
...         = Categorical(logits=scaled_logits)
...         last_char = m.sample()
...         generated_str += str(char_array[last_char])
...
...     return generated_str
```

Şimdi yeni bir metin oluşturalım:

```
>>> torch.manual_seed(1)
>> print(sample(model, starting_str='Ada'))
Ada yapılmıştı
ve ovylore'u düşünüyor musunuz, kaptan?" diye sordu Neb; "D ü ş ü n ü y o r u z . "

Bulundu, çıkarma zamanı geldi. Bu sinir u ç l a r ı  hakkında, belki de sona erdi?
Olabilir.
daha çok gül mü?"

"İleri!" diye haykırdı Pencroft, "onlar mıydı? Bana öyle geliyor

ki?" "Köpek Top--"
```

```
"Sventy ile ne mücadele edebilirdi ki?"

Pencroft, denizci ve Neb'in bu tenarvanın duygularını sınırladığını kanıtlayan
şeyleri denemek için zamanında kendilerini çağırdı ve sonra
hala saklanan kafası büyük bir öfkeyle Dorner'ı izliyor ne de onun tek
```

Gördüğünüz gibi, model çoğunlukla doğru kelimeler üretir ve bazı durumlarda cümleler kısmen anlamlıdır. Eğitim için girdi dizilerinin uzunluğu ve model mimarisi gibi eğitim parametrelerini daha fazla ayarlayabilirsiniz.

Ayrıca, üretilen örneklerin tahmin edilebilirliğini kontrol etmek için (yani, daha fazla rastgelelik eklemek yerine eğitim metninden öğrenilen kalıpları izleyerek metin oluşturmak), RNN modeli tarafından hesaplanan logitler örnekleme için Categorical'a aktarılmadan önce ölçeklendirilebilir. Ölçekleme faktörü, α, fizikteki sıcaklığın bir benzeri olarak yorumlanabilir. Daha yüksek sıcaklıklar, daha düşük sıcaklıklarda daha öngörülebilir davranışa karşı daha fazla entropi veya rastgelelik ile sonuçlanır. Logitlerin α α 1 ile ölçeklendirilmesiyle, softmax fonksiyonu

```
>>> logits = torch.tensor([[1.0, 1.0, 3.0]])
>> print('Ölçeklemeden önceki olasılıklar:        ',
...         nn.functional.softmax(logits, dim=1).numpy()[0])
>> print('0,5 ile ölçeklendirildikten sonra olasılıklar:',
...         nn.functional.softmax(0.5*logits, dim=1).numpy()[0])
>> print('0,1 ile ölçeklendirildikten sonra olasılıklar:',
...         nn.functional.softmax(0.1*logits, dim=1).numpy()[0])
Ölçeklemeden önceki olasılıklar:        [0.10650698 0.10650698 0 . 78698604]
Olasılıklar 0,5 ile ölçeklendirildikten sonra: [0,21194156 0 , 21194156 0 , 57611688]
Olasılıklar 0,1 ile ölçeklendirildikten sonra: [0,31042377 0 , 31042377 0 , 37915245]
```

tarafından hesaplanan olasılıklar, aşağıdaki kodda gösterildiği gibi daha düzgün hale gelir:

Gördüğünüz gibi, logitleri α α 1 ile ölçeklendirmek neredeyse tekdüze olasılıklarla sonuçlanır [0.31, 0.31, 0.38]. Şimdi, oluşturulan metni aşağıdaki noktalarda gösterildiği gibi α α $\alpha\alpha$ ve α α $\alpha\alpha\alpha$ ile karşılaştırabiliriz:

- α α $\alpha\alpha$ α $\alpha\alpha\alpha$ $\alpha\alpha\alpha\alpha\alpha\alpha\alpha\alpha\alpha\alpha$:

```
>>> torch.manual_seed(1)
>> print(sample(model, starting_str='Ada',
...                    scale_factor=2.0))
Ada kolonilerden biri mi?" diye sordu denizci, "Pasifik kıyılarına gelmek
mümkün değil."
"Evet," diye yanıtladı mühendis, "ve eğer ormanın konumu ve bataklık
yol söylenildiği gibi değilse, köpek kıyıya ilk çıkan değildi ve
kendilerini ağılda buldular.
```

 ilinpikbarbell@gmail.com

```
Yerleşimciler denizcinin hala denizin yüzeyinde olduğunu, deniz için
alınmadıklarını söylediler. Kıyı, Granit Ev'in pencerelerini inceleyebilecek
durumdaydı.
Denizci, denizciyi hor'a çevirdi.
```

- α α $\alpha\alpha$ α $\alpha\alpha\alpha\alpha$ $\alpha\alpha\alpha\alpha\alpha\alpha\alpha\alpha\alpha\alpha$:

```
>>> torch.manual_seed(1)
>> print(sample(model, starting_str='Ada',
...                    scale_factor=0.5))
Ada derin
incomele.
Manyl's', House, won's calcon-sglenderlessly," everful ineriorouins.,
pyra" into
Gerçek. Sometinivabes, iskumar gave-zen."

Bleshed but what cotch quadrap which little cedass
fell oprely
Andonem tarafından. Peditivall--"I dove Gurgeon. Onun için ne kararlıydı?

Küçük dönüşler, sahiplerinin planladığı çalılık inlerinden sonra geri
döndü; onlar
Biliyor musun? Whalerin şubesi I
pites, Dougg!-iteun," returnwe aid masses atong thoughts! Dak, Hem-
arches yone, Veay wantzer? Woblding,
Herbert, omep
```

Sonuçlar, logitlerin α α $\alpha\alpha$ ile ölçeklendirilmesinin (sıcaklığın artırılması) daha fazla rastgele metin ürettiğini göstermektedir. Üretilen metnin yeniliği ile doğruluğu arasında bir denge vardır.

Bu bölümde, bir diziden diziye (seq2seq) modelleme görevi olan karakter düzeyinde metin oluşturma ile çalıştık. Bu örnek kendi başına çok faydalı olmasa da, bu tür modeller için birkaç faydalı uygulama düşünmek kolaydır; örneğin, benzer bir RNN modeli, kullanıcılara basit sorgularda yardımcı olmak için bir sohbet robotu olarak eğitilebilir.

Özet

Bu bölümde, ilk olarak dizileri yapılandırılmış veriler veya görüntüler gibi diğer veri türlerinden farklı kılan özellikleri öğrendiniz. Daha sonra dizi modelleme için RNN'lerin temellerini ele aldık. Temel bir RNN modelinin nasıl çalıştığını öğrendiniz ve dizi verilerindeki uzun vadeli bağımlılıkları yakalama konusundaki sınırlamalarını tartıştınız. Daha sonra, temel RNN modellerinde yaygın olan patlayan ve kaybolan gradyan problemlerinin etkisini azaltmak için bir geçit mekanizmasından oluşan LSTM hücrelerini ele aldık.

6587191CA584E

RNN'lerin arkasındaki ana kavramları tartıştıktan sonra, PyTorch kullanarak farklı tekrarlayan katmanlara sahip birkaç RNN modeli uyguladık. Özellikle, duygu analizi için bir RNN modelinin yanı sıra metin oluşturmak için bir RNN modeli uyguladık.

Bir sonraki bölümde, bir RNN'yi çeviri görevlerinde uzun menzilli bağımlılıkları modellemesine yardımcı olan bir dikkat mekanizması ile nasıl güçlendirebileceğimizi göreceğiz. Ardından, son zamanlarda doğal dil işleme alanında teknolojinin geldiği noktayı daha da ileriye taşımak için kullanılan *transformer* adlı yeni bir derin öğrenme mimarisini tanıtacağız.

Kitabımızın Discord alanına katılın

Benzer düşünen insanlarla tanışmak ve 2000'den fazla üyemizle birlikte öğrenmek için Discord topluluğumuza katılın:
https://packt.link/MLwPyTorch

olimpikbarbell@gmail.com

16

Transformers - Dikkat Mekanizmaları ile Doğal Dil İşlemenin Geliştirilmesi

Bir önceki bölümde, bir duygu analizi projesi aracılığıyla **tekrarlayan sinir ağları** (**RNN'ler**) ve bunların **doğal dil işleme** (**NLP**) alanındaki uygulamaları hakkında bilgi edinmiştik. Ancak, son zamanlarda çeşitli NLP görevlerinde RNN tabanlı **diziden diziye** (**seq2seq**) modellerden daha iyi performans gösterdiği gösterilen yeni bir mimari ortaya çıkmıştır. Bu sözde **dönüştürücü** mimarisidir.

Dönüştürücüler doğal dil işlemede devrim yaratmış ve otomatik dil çevirisinden (https://ai.googleblog. com/2020/06/recent-advances-in-google-translate.html) protein dizilerinin temel özelliklerini modellemeye (https://www.pnas.org/content/118/15/e2016239118.short) ve insanların kod yazmasına yardımcı olan bir yapay zeka yaratmaya (https://github.blog/2021-06-29-introducing-github-copilot- ai-pair-programmer) kadar birçok etkileyici uygulamanın ön saflarında yer almıştır.

Bu bölümde, *dikkat* ve *öz-dikkatin* temel mekanizmalarını öğrenecek ve bunların orijinal transformatör mimarisinde nasıl kullanıldığını göreceksiniz. Ardından, dönüştürücülerin nasıl çalıştığını anlayarak, bu mimariden ortaya çıkan en etkili NLP modellerinden bazılarını keşfedecek ve PyTorch'ta BERT modeli olarak adlandırılan büyük ölçekli bir dil modelinin nasıl kullanılacağını öğreneceğiz.

Aşağıdaki konuları ele alacağız:

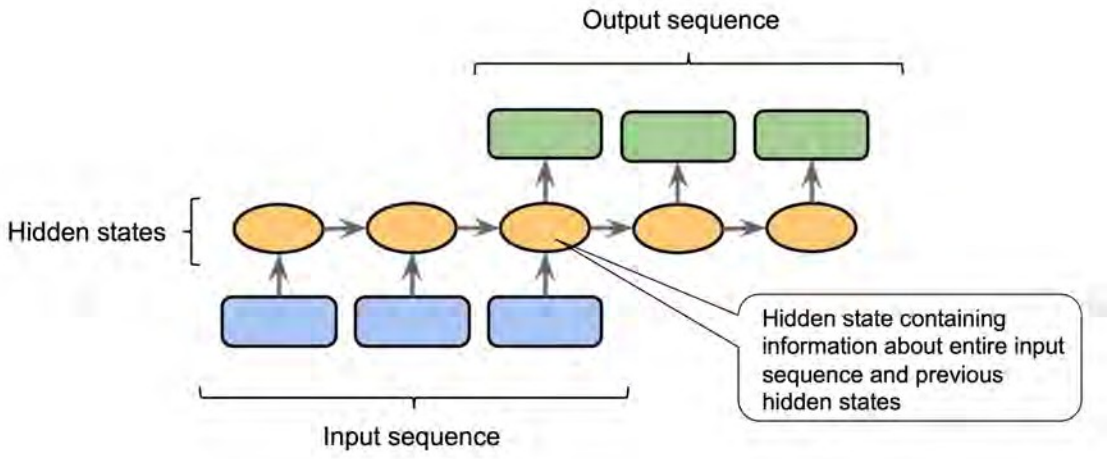
- Dikkat mekanizması ile RNN'lerin iyileştirilmesi
- Bağımsız öz dikkat mekanizmasının tanıtılması
- Orijinal transformatör mimarisinin anlaşılması
- Dönüştürücü tabanlı büyük ölçekli dil modellerinin karşılaştırılması
- Duygu sınıflandırması için BERT'e ince ayar yapma

RNN'lere bir dikkat mekanizması ekleme

Bu bölümde, tahmin modellerinin girdi dizisinin belirli kısımlarına diğerlerinden daha fazla odaklanmasına yardımcı olan bir **dikkat mekanizmasının** geliştirilmesinin arkasındaki motivasyonu ve bunun ilk olarak RNN'ler bağlamında nasıl kullanıldığını tartışıyoruz. Bu bölümün dikkat mekanizmasının neden geliştirildiğini açıklayan tarihsel bir perspektif sunduğunu unutmayın. Tek tek matematiksel ayrıntılar karmaşık görünüyorsa, bu bölümün odak noktası olan dönüştürücüler için kendi kendine dikkat mekanizmasını açıklayan bir sonraki bölüm için gerekli olmadıklarından, bunları atlamaktan çekinmeyin.

Dikkat, RNN'lerin bilgiye erişmesine yardımcı olur

Bir dikkat mekanizmasının gelişimini anlamak için, *Şekil 16.1*'de gösterildiği gibi, çeviriyi üretmeden önce tüm girdi dizisini (örneğin, bir veya daha fazla cümle) ayrıştıran dil çevirisi gibi bir **seq2seq görevi** için geleneksel RNN modelini düşünün:



Şekil 16.1: Seq2seq modelleme görevi için geleneksel bir RNN kodlayıcı-kod çözücü mimarisi

 alinpikbarbell@gmail.com

RNN neden ilk çıktıyı üretmeden önce tüm girdi cümlesini ayrıştırıyor? Bu, *Şekil 16.2'*de gösterildiği gibi, bir cümleyi kelime kelime çevirmenin muhtemelen dilbilgisi hatalarıyla sonuçlanacağı gerçeğiyle motive edilir:

Şekil 16.2: Bir cümleyi kelime kelime çevirmek dilbilgisi hatalarına yol açabilir

Bununla birlikte, *Şekil 16.2'*de gösterildiği gibi, bu seq2seq yaklaşımının bir sınırlaması, RNN'nin tüm giriş dizisini çevirmeden önce tek bir gizli birim aracılığıyla hatırlamaya çalışmasıdır. Tüm bilgilerin tek bir gizli birime sıkıştırılması, özellikle uzun diziler için bilgi kaybına neden olabilir. Bu nedenle, insanların cümleleri nasıl çevirdiğine benzer şekilde, her zaman adımında tüm girdi dizisine erişmek faydalı olabilir.

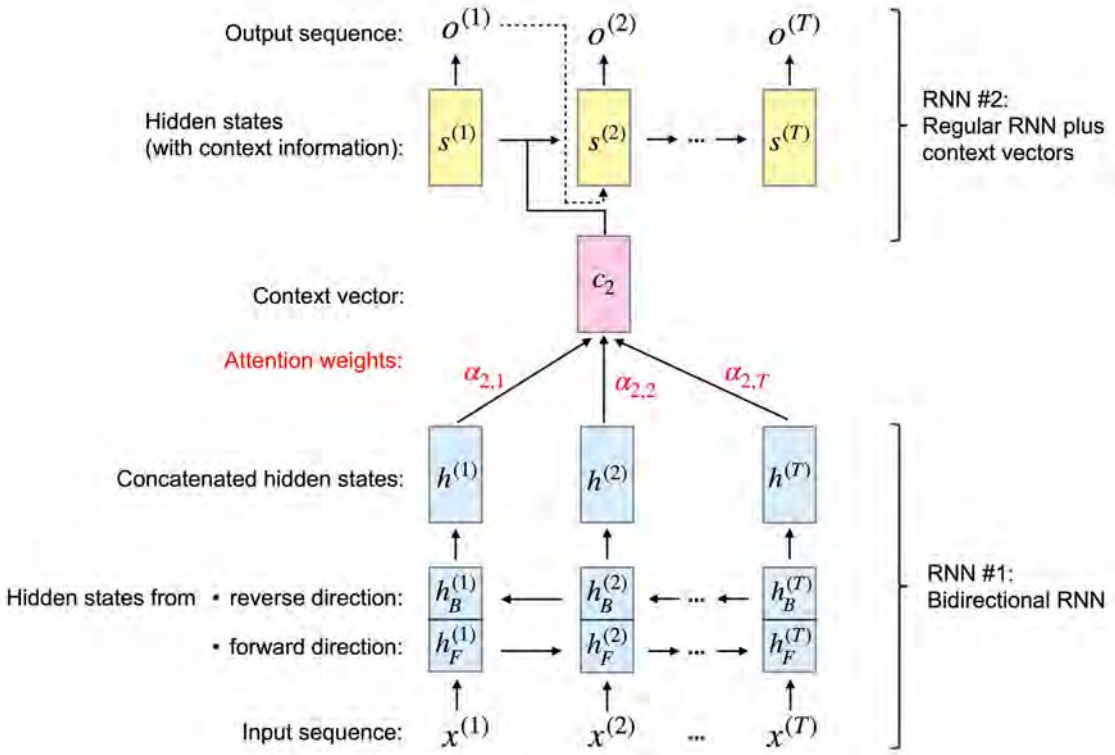
Normal bir RNN'nin aksine, bir dikkat mekanizması RNN'nin verilen her zaman adımında tüm girdi öğelerine erişmesine izin verir. Ancak, her zaman adımında tüm girdi dizisi öğelerine erişmek çok zor olabilir. Bu nedenle, RNN'nin girdi dizisinin en alakalı öğelerine odaklanmasına yardımcı olmak için, dikkat mekanizması her girdi öğesine farklı dikkat ağırlıkları atar. Bu dikkat ağırlıkları, belirli bir girdi dizisi öğesinin belirli bir zaman adımında ne kadar önemli veya ilgili olduğunu belirler. Örneğin, *Şekil 16.2'ye* tekrar dönersek, "mir, helfen, zu" kelimeleri "help" çıktı kelimesini üretmek için "kannst, du, Satz" kelimelerinden daha alakalı olabilir.

Bir sonraki alt bölümde, dil çevirisi için uzun dizilerin işlenmesine yardımcı olmak üzere bir dikkat mekanizması ile donatılmış bir RNN mimarisi tanıtılmaktadır.

RNN'ler için orijinal dikkat mekanizması

Bu alt bölümde, orijinal olarak dil çevirisi için geliştirilen ve ilk olarak aşağıdaki makalede ortaya çıkan dikkat mekanizmasının mekaniğini özetleyeceğiz: *Bahdanau, D., Cho, K. ve Bengio, Y.,* 2014, https:// arxiv.org/abs/1409.0473 tarafından *Hizalamayı ve Çevirmeyi Birlikte Öğrenerek Nöral Makine Çevirisi*.

Bir girdi dizisi $x\ x\ x\ ,\ x^{(1)(2)}\ ,\ldots\ ,\ x^{(T)}\ )$ verildiğinde, dikkat mekanizması her bir $x^{(i)}$ öğesine (veya daha spesifik olmak gerekirse, gizli temsiline) bir ağırlık atar ve modelin girdinin hangi kısmına odaklanması gerektiğini belirlemesine yardımcı olur. Örneğin, girdimizin bir cümle olduğunu ve daha büyük ağırlığa sahip bir kelimenin tüm cümleyi anlamamıza daha fazla katkıda bulunduğunu varsayalım. *Şekil 16.3'te* gösterilen dikkat mekanizmasına sahip RNN (daha önce bahsedilen makaleden sonra modellenmiştir) ikinci çıktı kelimesinin üretilmesine ilişkin genel kavramı göstermektedir:



Şekil 16.3: Dikkat mekanizmalı RNN

Şekilde gösterilen dikkat tabanlı mimari, sonraki alt bölümlerde açıklayacağımız iki RNN modelinden oluşmaktadır.

 olimpikbarbell@gmail.com

Çift yönlü bir RNN kullanarak girdilerin işlenmesi

Şekil 16.3'teki dikkat tabanlı RNN'nin ilk RNN'si (RNN #1) bağlam vektörleri üreten çift yönlü bir RNN'dir, c_i . Bir bağlam vektörünü girdi vektörünün artırılmış bir versiyonu olarak düşünebilirsiniz, $x^{(i)}$. Başka bir deyişle, c_i girdi vektörü, bir dikkat mekanizması aracılığıyla diğer tüm girdi öğelerinden gelen bilgileri de içerir. *Şekil 16.3'te görebileceğimiz* gibi, RNN #2 daha sonra çıktıları oluşturmak için RNN #1 tarafından hazırlanan bu bağlam vektörünü kullanır. Bu alt bölümün geri kalanında, RNN #1'in nasıl çalıştığını tartışacağız ve bir sonraki alt bölümde RNN #2'yi tekrar ziyaret edeceğiz.

Çift yönlü RNN #1, x giriş dizisini hem normal ileri yönde (1. . . T) hem de geri yönde ($TTT\ T1$) işler. Bir diziyi geriye doğru ayrıştırmak, orijinal girdi dizisini tersine çevirmekle aynı etkiye sahiptir - bir cümleyi ters sırada okumak gibi düşünün. Bunun arkasındaki mantık, mevcut girdilerin bir cümlede kendisinden önce veya sonra gelen dizi öğelerine veya her ikisine de bağlı olabileceğinden ek bilgi yakalamaktır.

Sonuç olarak, giriş dizisini iki kez (yani ileri ve geri) okuduğumuzda, her giriş dizisi öğesi için iki gizli durumumuz olur. Örneğin, ikinci girdi dizisi öğesi için
$x^{(2)}$, ileri geçişten $h^{(2)}_F$ gizli durumunu ve geriye doğru geçişten $h^{(2)}_B$ gizli durumunu elde ederiz. geçer. Bu iki gizli durum daha sonra $h^{(2)}$ gizli durumunu oluşturmak üzere birleştirilir. Örneğin, eğer
hem $h^{(2)}_F$ hem de $h^{(2)}_B$ 128 boyutlu vektörlerdir, birleştirilmiş gizli durum $h^{(2)}$ 256'dan oluşacaktır öğeler. Bu birleştirilmiş gizli durumu, her iki yönde de *j. kelimenin* bilgisini içerdiği için kaynak kelimenin "ek açıklaması" olarak düşünebiliriz.

Bir sonraki bölümde, bu birleştirilmiş gizli durumların ikinci RNN tarafından çıktıları oluşturmak için nasıl işlendiğini ve kullanıldığını göreceğiz.

Bağlam vektörlerinden çıktılar üretme

*Şekil 16.3'*te, RNN #2'yi çıktıları üreten ana RNN olarak düşünebiliriz. Gizli durumlara ek olarak, girdi olarak sözde bağlam vektörleri alır. Bir bağlam vektörü c_i , önceki alt bölümde RNN #1'den elde ettiğimiz birleştirilmiş gizli durumların, $h^{(1)} \ldots h^{(T)}$, ağırlıklandırılmış bir versiyonudur. *Birinci* girdinin bağlam vektörünü ağırlıklı bir toplam olarak hesaplayabiliriz:

$$c_i = \sum_{j=1}^{T} \alpha_{ij} h^{(j)}$$

Burada, α_{ij}, $j = 1, \ldots, T$ giriş dizisi üzerindeki dikkat ağırlıklarını birinci giriş dizisi elemanı bağlamında temsil eder. Her *bir girdi dizisi elemanının* benzersiz bir dikkat ağırlıkları kümesine sahip olduğunu unutmayın. Dikkat ağırlıklarının α_{ij} hesaplanmasını bir sonraki alt bölümde tartışacağız.

Bu alt bölümün geri kalanında, bağlam vektörlerinin önceki şekildeki ikinci RNN (RNN #2) aracılığıyla nasıl kullanıldığını tartışalım. Tıpkı vanilya (normal) bir RNN gibi, RNN #2 de gizli durumlar kullanır. Yukarıda bahsedilen "açıklama" ile nihai çıktı arasındaki gizli katmanı göz önünde bulundurarak, i zamanındaki gizli durumu $s^{(i)}$ olarak gösterelim. Şimdi, RNN #2 yukarıda bahsedilen bağlam vektörünü alır

6587191CA584E

c_i her i. zaman adımında girdi olarak.

 alinpikbarbell@gmail.com

Şekil 16.3'te, $s^{(ii}$ gizli durumunun, i zamanında $y^{(ii}$ hedef kelimesi için tahmin edilen $o^{(ii}$ çıktısını oluşturmak için kullanılan önceki $s^{(ii1i}$ gizli durumuna, önceki $y^{(ii1i}$ hedef kelimesine ve $c^{(ii}$ bağlam vektörüne bağlı olduğunu gördük. y dizi vektörünün, eğitim sırasında mevcut olan xx girdi dizisinin doğru çevirisini temsil eden dizi vektörünü ifade ettiğini unutmayın. Eğitim sırasında, gerçek etiket (kelime) $y^{(ii}$ bir sonraki duruma $s^{(ii1i}$ beslenir; bu gerçek etiket bilgisi tahmin (çıkarım) için mevcut olmadığından, bunun yerine önceki şekilde gösterildiği gibi tahmin edilen çıktı o beslenir.$^{(ii}$

Yukarıda tartıştıklarımızı özetlemek gerekirse, dikkat tabanlı RNN iki RNN'den oluşur. RNN #1 giriş dizisi elemanlarından bağlam vektörleri hazırlar ve RNN #2 bağlam vektörlerini giriş olarak alır. Bağlam vektörleri, ağırlıklar α_{ii} dikkat ağırlıkları olmak üzere girdiler üzerinde ağırlıklı bir toplam yoluyla hesaplanır. Bir sonraki alt bölümde bu dikkat ağırlıklarını nasıl hesapladığımız tartışılmaktadır.

Dikkat ağırlıklarının hesaplanması

Son olarak, bulmacamızdaki son eksik parçayı ziyaret edelim - dikkat ağırlıkları. Bu ağırlıklar girdileri (ek açıklamalar) ve çıktıları (bağlamlar) çift yönlü olarak birbirine bağladığından, her dikkat ağırlığının α_{ii} iki alt simgesi vardır: j girdinin indeks konumunu ve i çıktı indeks konumunu ifade eder. Dikkat ağırlığı α_{ii}, e_{ii} hizalama puanının normalleştirilmiş bir versiyonudur; burada hizalama puanı, j konumundaki girdinin i konumundaki çıktı ile ne kadar iyi eşleştiğini değerlendirir. Daha açık olmak gerekirse, dikkat ağırlığı hizalama puanlarının aşağıdaki gibi normalleştirilmesiyle hesaplanır:

$$\alpha_{ii} \equiv \frac{\exp(e_{ii})}{\sum_{iik1}^{T} \exp(e_{ii})}$$

Bu *denklemin*, *Bölüm 12, PyTorch ile Sinir Ağı Eğitiminin Paralelleştirilmesi*, *soßmax fonksiyonu aracılığıyla çok sınıflı sınıflandırmada sınıf olasılıklarının tahmin edilmesi* bölümünde tartıştığımız softmax fonksiyonuna benzer olduğunu unutmayın. Sonuç olarak, dikkat ağırlıkları $\alpha_{i1} \ldots \alpha_{iT}$ toplamı 1'dir.

Şimdi, özetlemek gerekirse, dikkat tabanlı RNN modelini üç kısma ayırabiliriz. İlk kısım girdinin çift yönlü açıklamalarını hesaplar. İkinci kısım, orijinal girdi yerine bağlam vektörlerini kullanması dışında orijinal RNN'ye çok benzeyen tekrarlayan bloktan oluşur. Son bölüm, her bir girdi ve çıktı öğesi çifti arasındaki ilişkiyi tanımlayan dikkat ağırlıklarının ve bağlam vektörlerinin hesaplanmasıyla ilgilidir.

Transformatör mimarisi de bir dikkat mekanizması kullanır, ancak dikkat tabanlı RNN'den farklı olarak, yalnızca **kendi kendine dikkat** mekanizmasına dayanır ve RNN'de bulunan tekrarlayan süreci içermez. Başka bir deyişle, bir dönüştürücü modeli, diziyi her seferinde bir öğe okumak ve işlemek yerine tüm girdi dizisini bir kerede işler. Bir sonraki bölümde, dönüştürücü mimarisini daha ayrıntılı olarak tartışmadan önce, kendi kendine dikkat mekanizmasının temel bir biçimini tanıtacağız.

Öz-dikkat mekanizmasının tanıtılması

Önceki bölümde, dikkat mekanizmalarının uzun dizilerle çalışırken bağlamı hatırlama konusunda

6587191CA584E

RNN'lere yardımcı olabileceğini gördük. Bir sonraki bölümde göreceğimiz gibi, bir RNN'nin tekrarlayan kısımları olmadan tamamen dikkat temelli bir mimariye sahip olabiliriz. Bu dikkat tabanlı mimari **transformatör** olarak bilinir ve daha sonra daha ayrıntılı olarak tartışacağız.

alinpikbarbell@gmail.com

Aslında transformatörler ilk bakışta biraz karmaşık görünebilir. Bu nedenle, bir sonraki bölümde transformatörleri tartışmadan önce, transformatörlerde kullanılan öz-dikkat mekanizmasına bir göz **atalım.** Aslında, göreceğimiz gibi, bu öz-dikkat mekanizması bir önceki bölümde tartıştığımız dikkat mekanizmasının farklı bir çeşididir. Daha önce tartışılan dikkat mekanizmasını iki farklı modülü, yani RNN'nin kodlayıcı ve kod çözücüsünü birbirine bağlayan bir işlem olarak düşünebiliriz. Göreceğimiz gibi, öz dikkat sadece girdiye odaklanır ve sadece girdi unsurları arasındaki bağımlılıkları yakalar. iki modülü birbirine bağlamadan.

İlk alt bölümde, herhangi bir öğrenme parametresi içermeyen ve girdiye yönelik bir ön işleme adımına çok benzeyen temel bir öz dikkat biçimini tanıtacağız. Daha sonra ikinci alt bölümde, transformatör mimarisinde kullanılan ve öğrenilebilir parametreler içeren öz dikkatin yaygın versiyonunu tanıtacağız.

Kendine dikkat etmenin temel bir biçimiyle başlamak

Öz dikkati tanıtmak için, T uzunluğunda bir girdi dizimiz olduğunu varsayalım, $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}$, ve bir çıktı dizisi, $z^{(1)}, z^{(2)}, \ldots, z^{(T)}$. Karışıklığı önlemek için, o'yi tüm transformatör modelinin nihai çıktısı olarak ve z'yi modelde bir ara adım olduğu için öz dikkat katmanının çıktısı olarak kullanacağız.

Bu dizilerdeki her bir i. eleman, $\boldsymbol{x}^{(i)}$ ve $z^{(i)}$, d boyutunda vektörlerdir (yani, $\boldsymbol{x}^{(i)} \in R^d$) ve RNN'lere benzer şekilde i. konumdaki girdi için özellik bilgisini temsil eder. Daha sonra, bir seq2seq görevi için, öz dikkatin amacı, mevcut girdi elemanının diğer tüm girdi elemanlarına olan bağımlılıklarını modellemektir. Bunu başarmak için, öz dikkat mekanizmaları üç aşamadan oluşur. İlk olarak, mevcut öğe ile dizideki diğer tüm öğeler arasındaki benzerliğe dayalı olarak önem ağırlıkları türetiyoruz. İkinci olarak, ağırlıkları normalleştiririz, bu da genellikle zaten bilinen softmax fonksiyonunun kullanımını içerir. Üçüncü olarak, dikkat değerini hesaplamak için bu ağırlıkları ilgili dizi öğeleriyle birlikte kullanırız.

Daha resmi olarak, öz dikkatin çıktısı, $z^{(i)}$, tüm T giriş dizilerinin ağırlıklı toplamıdır, $\boldsymbol{x}^{(j)}$ (burada $jj\ j\ 1jj\ jT$). Örneğin, *birinci* giriş elemanı için karşılık gelen çıkış değeri aşağıdaki gibi hesaplanır:

$$z(i) = \sum_{iij1}^{T} \alpha ii^{\boldsymbol{x}(ii)}$$

Dolayısıyla, $z^{(i)}$'yi, $\boldsymbol{x}^{(i)}$ girdi vektöründe, ilgili dikkat ağırlıkları ile ağırlıklandırılmış diğer tüm girdi dizisi öğelerini içeren, bağlama duyarlı bir gömme vektörü olarak düşünebiliriz. Burada, dikkat ağırlıkları, αii, mevcut giriş elemanı, $\boldsymbol{x}^{(i)}$, ve giriş dizisindeki diğer tüm elemanlar, $\boldsymbol{x}^{(1)} \ldots$ arasındaki benzerliğe göre hesaplanır. $\boldsymbol{x}^{(T)}$. Daha somut olarak, bu benzerlik sonraki paragraflarda açıklanan iki adımda hesaplanır.

İlk olarak, mevcut giriş elemanı $\boldsymbol{x}^{(i)}$ ile giriş dizisindeki başka bir eleman $\boldsymbol{x}^{(jj)}$ arasındaki nokta çarpımını hesaplarız:

$$\omega ii = \boldsymbol{x}^{(i)^{\mathrm{T}}} \boldsymbol{x}^{(ii)}$$

Dikkat ağırlıklarını (a_{iii}) elde etmek için ω_{ii} değerlerini normalize etmeden önce, ω_{ii} değerlerini nasıl hesapladığımızı bir kod örneği ile açıklayalım. Burada, bir giriş cümlemiz olduğunu varsayalım "bana yardım edebilir misin
Bu cümleyi çevirin" ifadesi, *Bölüm 15, Tekrarlayan Sinir Ağları Kullanarak Sıralı Verilerin Modellenmesi'nde* açıklandığı gibi bir sözlük aracılığıyla bir tamsayı gösterimine eşlenmiştir:

```
>> import torch
>> cümle = torch.tensor(
>>>      [0, # olabilir
>>>       7, # sen
>>>       1, # Yardım
>>>       2, # Ben
>>>       5, # için
>>>       6, # tercüme
                etmek
>>>       4, # Bu
>>>       3] # cümle
>>> )

>>> cümle
tensör([0, 7, 1, 2, 5, 6, 4, 3])
```

Ayrıca bu cümleyi bir gömme katmanı aracılığıyla gerçek sayı vektör gösterimine kodladığımızı varsayalım. Burada, gömme boyutumuz 16'dır ve sözlük boyutunun 10 olduğunu varsayıyoruz. Aşağıdaki kod sekiz kelimemizin kelime gömülerini üretecektir:

```
>>> torch.manual_seed(123)
>>> embed = torch.nn.Embedding(10, 16)
>>> embedded_sentence = embed(sentence).detach()
>> embedded_sentence.shape
torch.Size([8, 16])
```

Şimdi, ω_{ii}'yi ith ve *j*. kelime gömüleri arasındaki nokta çarpımı olarak hesaplayabiliriz. Bunu tüm ω_{ii} değerleri için aşağıdaki gibi yapabiliriz:

```
>>> omega = torch.empty(8, 8)
>> for i, x_i in enumerate(embedded_sentence):
>>>      for j, x_j in enumerate(embedded_sentence):
>>>          omega[i, j] = torch.dot(x_i, x_j)
```

Önceki kodun okunması ve anlaşılması kolay olsa da, for döngüleri çok verimsiz olabilir, bu nedenle bunun yerine matris çarpımını kullanarak hesaplayalım:

```
>> omega_mat = gömülü_sentence.matmul(gömülü_sentence.T)
```

olimpikbarbell@gmail.com

Bu matris çarpımının beklenen sonuçları üretip üretmediğini kontrol etmek için torch.allclose fonksiyonunu kullanabiliriz. İki tensör aynı değerleri içeriyorsa, burada görebileceğimiz gibi

```
>> torch.allclose(omega_mat, o m e g a )
True
```

torch.allclose True döndürür:

Birinci girdi ve dizideki tüm girdiler için benzerliğe dayalı ağırlıkların ($\boldsymbol{x}^{(1)}$ ila $\boldsymbol{x}^{(T)}$), "ham" ağırlıkların (ω_{i1} ila ω_{iT}) nasıl hesaplanacağını öğrendik. Dikkat ağırlıklarını, α_{iii}, ω_{iii} değerlerini bilinen softmax fonksiyonu ile normalize ederek aşağıdaki gibi elde edebiliriz:

$$\alpha_{ii} = \frac{\exp(\omega_{ii})}{\sum_{iij1}^{T}\exp(\omega_{ii})} = \text{softmax}\left([\omega_{iii}]_{iij1jT}\right)$$

Paydanın tüm girdi elemanları (1. . . T) üzerinde bir toplam içerdiğine dikkat edin. Dolayısıyla, bu softmax fonksiyonunun uygulanması nedeniyle, ağırlıklar bu normalleştirmeden sonra 1'e toplanacaktır, yani,

$$\sum_{iij1}^{T} \alpha_{ii} = 1$$

PyTorch'un softmax fonksiyonunu kullanarak dikkat ağırlıklarını aşağıdaki gibi hesaplayabiliriz:

```
>> import torch.nn.functional as F
>> dikkat_ağırlıkları = F .softmax(omega, dim=1)
>> attention_weights.shape
torch.Size([8, 8])
```

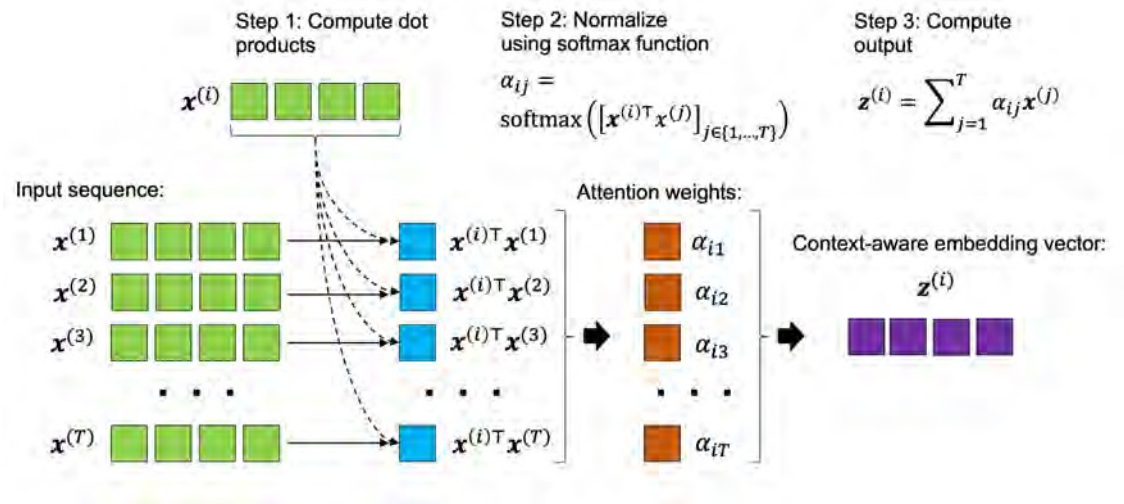
attention_weights'ın 8×8'lik bir matris olduğunu ve her bir öğenin bir dikkat ağırlığını temsil ettiğini unutmayın,
α_{ii}. Örneğin, *i.* girdi kelimesini işliyorsak, bu matrisin i. satırı cümledeki tüm kelimeler için karşılık gelen dikkat ağırlıklarını içerir. Bu dikkat ağırlıkları, her bir kelimenin i. kelime ile ne kadar alakalı olduğunu gösterir. Dolayısıyla, bu dikkat matrisindeki sütunların toplamı 1 olmalıdır, Bunu aşağıdaki kod ile teyit edebiliriz:

```
>>> attention_weights.sum(dim=1)
tensör([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000])
```

Şimdi dikkat ağırlıklarının nasıl hesaplanacağını gördüğümüze göre, öz dikkat işleminin arkasındaki üç ana adımı özetleyelim:

1. Verilen bir giriş elemanı, $\boldsymbol{x}^{(i)}$ ve {1, ..., *T*} kümesindeki her *j. eleman* için nokta çarpımını hesaplayın, $\boldsymbol{x}^{(i)\,T}\boldsymbol{x}^{(jj)}$
2. Softmax fonksiyonunu kullanarak nokta çarpımlarını normalleştirerek dikkat ağırlığını, α_{ii}, elde edin
3. Çıktıyı, $z^{(i)}$, tüm giriş dizisi üzerinden ağırlıklı toplam olarak hesaplayın: $z^{(i)} = \sum_{iij1}^{T} \alpha_{ii}\boldsymbol{x}^{(ii)}$

Bu adımlar *Şekil 16.4'te* daha ayrıntılı olarak gösterilmiştir:



Şekil 16.4: Örnekleme amaçlı temel bir öz dikkat süreci

Son olarak, bağlam vektörlerini, $z^{(i)}$, girdilerin dikkat ağırlıklı toplamı olarak hesaplamak için bir kod örneği görelim (*Şekil 16.4'*teki 3. adım). Özellikle, ikinci girdi kelimesi için bağlam vektörünü

```
>> x_2 = gömülü_sentence[1, :]
>> context_vec_2 = torch.zeros(x_2.shape)
>> for j  in range(8):
...         x_j = gömülü_sentence[j, :]
...         context_vec_2 += attention_weights[1, j] * x_j
>> context_vec_2
tensör([-9.3975e-01, -4.6856e-01, 1 . 0311e+00, -2.8192e-01, 4 . 9373e-01,
-1.2896e-02, -2.7327e-01, -7.6358e-01, 1 . 3958e+00, -9.9543e-01,
-7.1288e-04, 1 . 2449e+00, - 7 . 8077e-02, 1 . 2765e+00, -1.4589e+00,
-2.1601e+00])
```

hesapladığımızı varsayalım, yani $z^{(2)}$:

 alinpikbarbell@gmail.com

Yine, matris çarpımı kullanarak bunu daha verimli bir şekilde başarabiliriz. Aşağıdaki kodu kullanarak, sekiz girdi sözcüğünün tümü için bağlam vektörlerini hesaplıyoruz:

```
>> context_vectors = torch.matmul(
...         attention_weights, embedded_sentence)
```

embedded_sentence içinde saklanan girdi kelime gömülerine benzer şekilde, context_vectors matrisi 8×16 boyutuna sahiptir. Bu matristeki ikinci satır, ikinci girdi kelimesi için bağlam vektörünü içerir ve torch.allclose() kullanarak uygulamayı tekrar kontrol edebiliriz:

```
>> torch.allclose(context_vec_2, context_vectors[1])
True
```

Gördüğümüz gibi, ikinci bağlam vektörünün manuel for döngüsü ve matris hesaplamaları aynı sonuçları vermiştir.

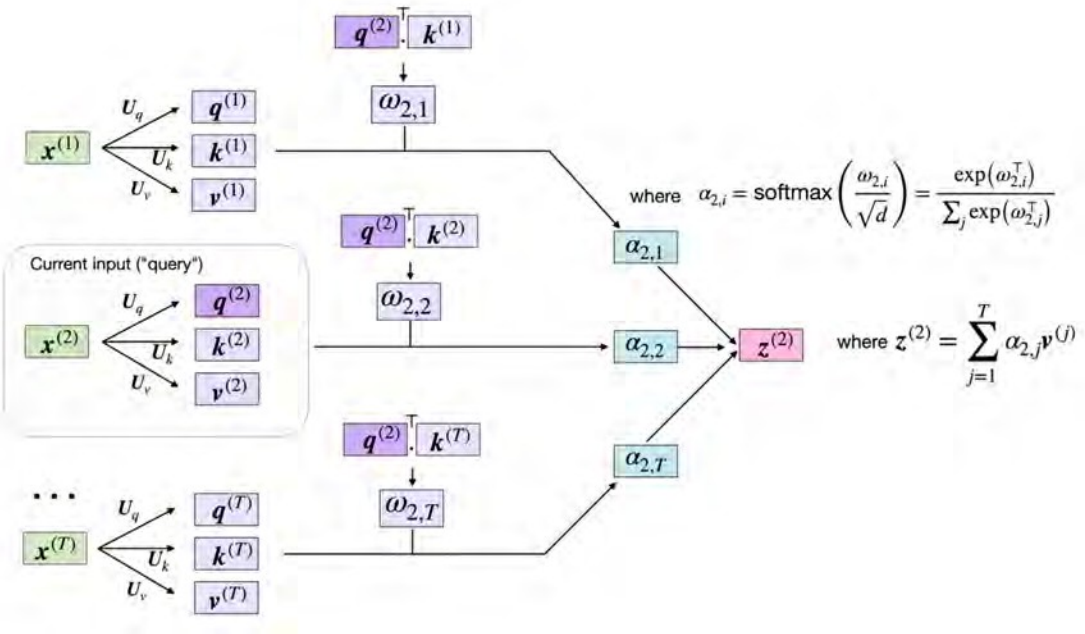
Bu bölümde öz dikkatin temel bir formunu uyguladık ve bir sonraki bölümde, sinir ağı eğitimi sırasında optimize edilebilen öğrenilebilir parametre matrisleri kullanarak bu uygulamayı değiştireceğiz.

Öz-dikkat mekanizmasının parametrelendirilmesi: ölçeklendirilmiş nokta-çarpım dikkati

Kendi kendine **dikkatin** arkasındaki temel kavramla tanıştığınıza göre, bu alt bölümde dönüştürücü mimarisinde kullanılan ölçeklendirilmiş **nokta-çarpım dikkati** adı verilen daha gelişmiş kendi kendine **dikkat** mekanizması özetlenmektedir. Önceki alt bölümde, çıktıları hesaplarken öğrenilebilir parametreleri dahil etmediğimizi unutmayın. Başka bir deyişle, daha önce tanıtılan temel öz dikkat mekanizmasını kullanan dönüştürücü modeli, belirli bir dizi için model optimizasyonu sırasında dikkat değerlerini nasıl güncelleyebileceği veya değiştirebileceği konusunda oldukça sınırlıdır. Öz dikkat mekanizmasını daha esnek ve model optimizasyonuna uygun hale getirmek için, model eğitimi sırasında model parametreleri olarak uydurulabilecek üç ek ağırlık matrisi tanıtacağız. Bu üç ağırlık matrisini U_q , U_{kk} ve U_v olarak gösteriyoruz. Bunlar, girdileri aşağıdaki gibi sorgu, anahtar ve değer dizisi öğelerine yansıtmak için kullanılır:

- **Sorgu dizisi**: $\boldsymbol{qq}^{(i)} = U_q \boldsymbol{xx}^{(i)}$ için i $i[1, T]$
- **Anahtar dizisi**: $\boldsymbol{kk}^{(i)} = {}_{Ukk} \boldsymbol{xx}^{(i)}$ için i $i[1, T]$
- **Değer dizisi**: $v^{(i)} = Uv\boldsymbol{xx}^{(i)}$ için i $i[1, T]$

Şekil 16.5, ikinci giriş elemanına karşılık gelen bağlama duyarlı em- yatak vektörünü hesaplamak için bu bireysel bileşenlerin nasıl kullanıldığını göstermektedir:



Şekil 16.5: İkinci dizi elemanının bağlama duyarlı gömme vektörünün hesaplanması

Sorgu, anahtar ve değer terminolojisi

Orijinal transformatör makalesinde kullanılan sorgu, anahtar ve değer terimleri bilgi erişim sistemleri ve veritabanları tarafından kullanılmaktadır. Örneğin, bir sorgu girdiğimizde, bu sorgu belirli değerlerin alındığı anahtar değerlerle eşleştirilir.

Burada, hem $\boldsymbol{qq}^{(i)}$ hem de $\boldsymbol{kk}^{(i)}$ d_{kk} boyutunda vektörlerdir. Bu nedenle, U_q ve U_{kk} projeksiyon matrisleri $d_{kk} \times d$ şekline sahipken, U_v $d_v \times d$ şekline sahiptir. (d'nin her bir kelime vektörünün boyutluluğu olduğunu unutmayın, $\boldsymbol{xx}^{(i)}$.) Basitlik için, bu vektörleri aynı şekle sahip olacak şekilde tasarlayabiliriz,

```
>>> torch.manual_seed(123)
>> d = embedded_sentence.shape[1]
>>> U_query = torch.rand(d, d)
>>> U_key = torch.rand(d, d)
>>> U_değeri = torch.rand(d, d)
```

örneğin $d_{kk} = d_v = d$. Kod aracılığıyla ek sezgi sağlamak için, bu projeksiyon matrislerini aşağıdaki gibi başlatabiliriz:

dilinpikbarbell@gmail.com

Sorgu projeksiyon matrisini kullanarak sorgu dizisini hesaplayabiliriz. Bu örnek için, *Şekil 16.5'te*

```
>> x_2 = gömülü_sentence[1]
>>> sorgu_2 = U_query.matmul(x_2)
```

gösterildiği gibi, ikinci girdi elemanını, $\boldsymbol{xx}^{(i)}$, sorgumuz olarak düşünün:

Benzer şekilde, anahtar ve değer dizilerini hesaplayabiliriz, $\boldsymbol{kk}^{(i)}$ ve $v^{(i)}$:

```
>>> anahtar_2 = U_key.matmul(x_2)
>>> değer_2 = U_değer.matmul(x_2)
```

Ancak, *Şekil 16.5'*ten de görebileceğimiz gibi, diğer tüm girdi öğeleri için anahtar ve değer dizilerine de ihtiyacımız vardır ve bunları aşağıdaki gibi hesaplayabiliriz:

```
>>> keys = U_key.matmul(embedded_sentence.T).T
>>> değerler = U_değer.matmul(gömülü_sentence.T).T
```

Anahtar matrisinde birinci satır, birinci girdi elemanının anahtar dizisine karşılık gelir ve aynı şey değer matrisi için de geçerlidir. Bunu, True döndürmesi gereken torch.allclose() işlevini tekrar kullanarak doğrulayabiliriz:

```
>>> keys = U_key.matmul(embedded_sentence.T).T
>>> torch.allclose(key_2, keys[1])
>>> değerler = U_değer.matmul(gömülü_sentence.T).T
>>> torch.allclose(değer_2, değerler[1])
```

Önceki bölümde, normalleştirilmemiş ağırlıkları, ω_{ii}, verilen giriş dizisi elemanı, $\boldsymbol{xx}^{(i)}$, ve j. dizi elemanı, $\boldsymbol{xx}^{(j)}$ arasındaki ikili nokta çarpımı olarak hesapladık. Şimdi, öz dikkatin bu parametreleştirilmiş versiyonunda, ω_{ii}'yi sorgu ve anahtar arasındaki nokta çarpımı olarak hesaplarız:

$$\omega = \boldsymbol{qq}^{(i)T}\boldsymbol{kk}^{(j)}$$

Örneğin, aşağıdaki kod normalleştirilmemiş dikkat ağırlığını, ω_{23} , yani sorgumuz ile üçüncü

```
>> omega_23 = query_2.dot(keys[2])
>> omega_23
tensörü(14.3667)
```

girdi dizisi öğesi arasındaki nokta çarpımını hesaplar:

Bunlara daha sonra ihtiyacımız olacağından, bu hesaplamayı tüm anahtarlara ölçeklendirebiliriz:

```
>>> omega_2 = query_2.matmul(keys.T)
>> omega_2
tensör([-25.1623,      9.3602, 14.3667, 32.1482, 53.8976, 46.6626, -1.2131,
-32.9391])
```

6587191CA584E

Öz dikkatte bir sonraki adım, softmax fonksiyonunu kullanarak normalize edilmemiş dikkat ağırlıklarından (ω_{ii}) normalize edilmiş dikkat ağırlıklarına (α_{ii}) geçmektir. Daha sonra, softmax fonksiyonu aracılığıyla normalleştirmeden önce ω_{ii}'yi ölçeklendirmek için aşağıdaki gibi $1/\sqrt{m}$ kullanabiliriz:

$$\alpha_{ii} = \text{softmax}\left(\frac{\omega_{ii}}{\sqrt{m}}\right)$$

Tipik olarak m m d_{kk} olan ω_{ii}'nin $1/\sqrt{m}$ ile ölçeklendirilmesinin, ağırlık vektörlerinin Öklid uzunluğunun yaklaşık olarak aynı aralıkta olmasını sağlayacağını unutmayın.

Aşağıdaki kod, sorgu olarak ikinci girdi öğesine göre tüm girdi dizisi için dikkat ağırlıklarını hesaplamak üzere bu normalleştirmeyi uygulamak içindir:

```
>>> attention_weights_2 = F.softmax(omega_2 / d**0.5, dim=0)
>> attention_weights_2
tensör([2.2317e-09, 1.2499e-05, 4.3696e-05, 3.7242e-03, 8.5596e-01, 1.4025e-01,
8.8896e-07, 3.1936e-10])
```

Son olarak, çıktı, değer dizilerinin ağırlıklı bir ortalamasıdır: $z^{(i)} = \sum_{ii=1}^{T} \alpha_{ii} v^{(ii)}$, uygulanabilir- aşağıdaki gibi yönlendirilmiştir:

```
>> context_vector_2 = attention_weights_2.matmul(values)
>> context_vector_2
tensör([-1.2226, -3.4387, -4.3928, -5.2125, -1.1249, -3.3041,
-1.4316, -3.2765, -2.5114, -2.6105, -1.5793, -2.8433, -2.4142,
-0.3998, -1.9917, -3.3499])
```

Bu bölümde, ilgili dikkat puanlarına göre ağırlıklandırılan tüm girdi öğelerini dahil ederek bağlama duyarlı gömme vektörleri oluşturmamızı sağlayan eğitilebilir parametrelere sahip bir öz dikkat mekanizması tanıttık. Bir sonraki bölümde, bu bölümde tanıtılan kendi kendine dikkat mekanizması etrafında merkezlenen bir sinir ağı mimarisi olan transformatör mimarisi hakkında bilgi edineceğiz.

İhtiyacımız olan tek şey dikkat: orijinal transformatör mimarisinin tanıtılması

İlginç bir şekilde, orijinal dönüştürücü mimarisi ilk olarak bir RNN'de kullanılan bir dikkat mekanizmasına dayanmaktadır. Başlangıçta, bir dikkat mekanizması kullanmanın ardındaki amaç, uzun cümlelerle çalışırken RNN'lerin metin oluşturma yeteneklerini geliştirmekti. Ancak, RNN'ler için dikkat mekanizmalarını denedikten sadece birkaç yıl sonra, araştırmacılar dikkat tabanlı bir dil modelinin tekrarlayan katmanlar silindiğinde daha da güçlü olduğunu keşfetti. Bu durum, bu bölümün ve geri kalan bölümlerin ana konusu olan **dönüştürücü mimarisinin** geliştirilmesine yol açmıştır.

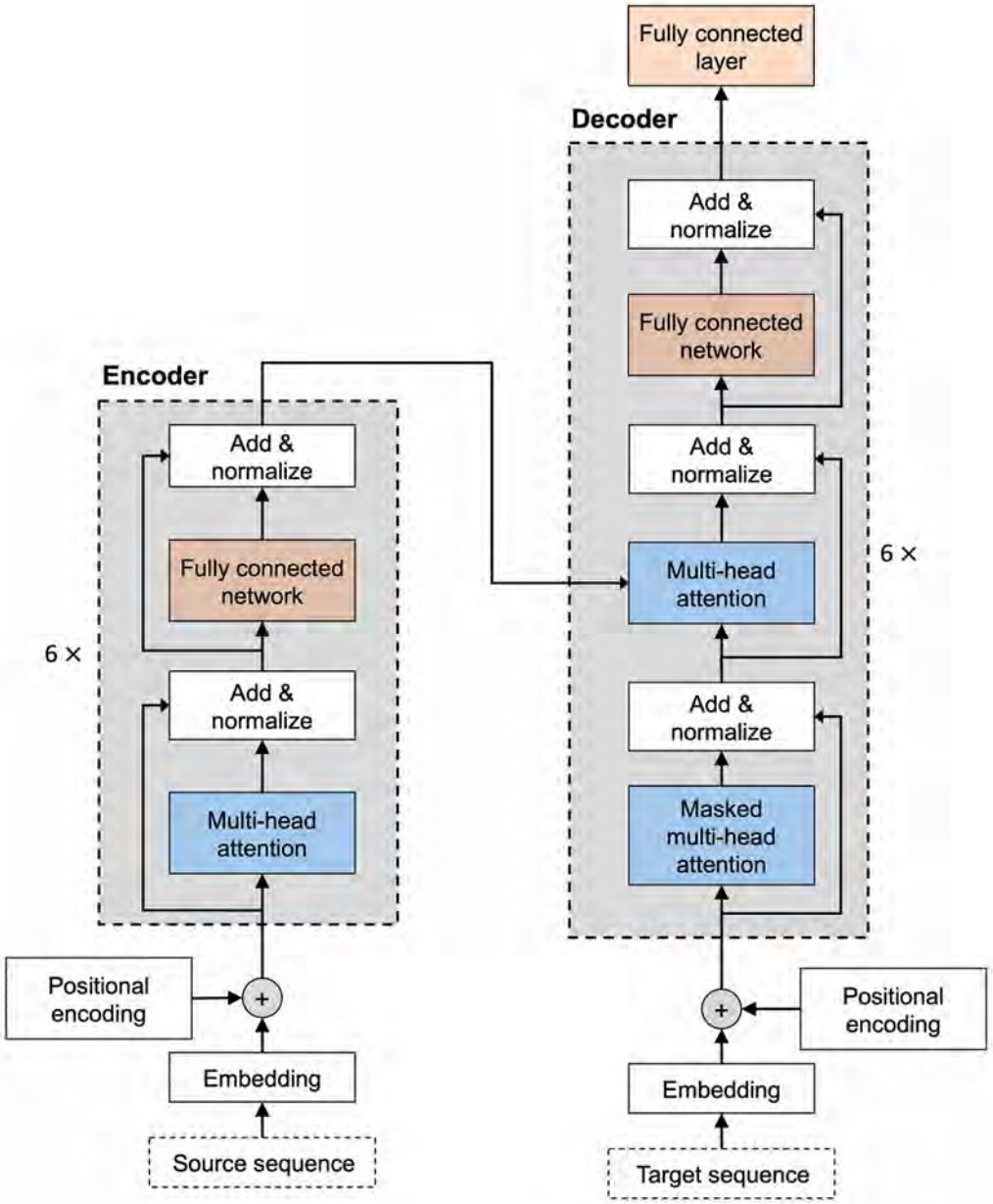
Dönüştürücü mimarisi ilk olarak *A. Vaswani* ve meslektaşları tarafından NeurIPS 2017'de *Attention Is All You Need adlı* makalede önerilmiştir (https://arxiv.org/abs/1706.03762). Öz-dikkat mekanizması sayesinde, bir dönüştürücü modeli bir girdi dizisindeki öğeler arasındaki uzun

alinpikbarbell@gmail.com

menzilli bağımlılıkları yakalayabilir - bir NLP bağlamında; örneğin, bu, modelin bir girdi cümlesinin anlamını daha iyi "anlamasına" yardımcı olur.

6587191CA584E

Bu dönüştürücü mimarisi başlangıçta dil çevirisi için tasarlanmış olsa da, İngilizce bileşen ayrıştırma, metin oluşturma ve metin sınıflandırma gibi diğer görevlere de genelleştirilebilir. Daha sonra, bu orijinal dönüştürücü mimarisinden türetilen BERT ve GPT gibi popüler dil modellerini tartışacağız. Orijinal dönüştürücü makalesinden uyarladığımız *Şekil 16.6*, bu bölümde tartışacağımız ana mimariyi ve bileşenleri göstermektedir:



Şekil 16.6: Orijinal transformatör mimarisi

 olimpikbarbell@gmail.com

Aşağıdaki alt bölümlerde, bu orijinal dönüştürücü modelini iki ana bloğa ayırarak adım adım inceleyeceğiz: bir kodlayıcı ve bir kod çözücü. Kodlayıcı orijinal sıralı girdiyi alır ve çok kafalı bir öz dikkat modülü kullanarak katıştırmaları kodlar. Kod çözücü işlenmiş girdiyi alır ve *maskelenmiş* bir öz dikkat biçimi kullanarak ortaya çıkan diziyi (örneğin, çevrilmiş cümle) çıkarır.

Çok kafalı dikkat yoluyla bağlam katıştırmalarını kodlama

Kodlayıcı bloğunun genel amacı, $XX\ X\ XXX^{(1)}$, $XX^{(2)}$, … , $XX^{(T)}$) sıralı bir girdi almak ve daha sonra kod çözücüye aktarılacak olan $Z\ Z\ Zz^{(1)}$, $z^{(2)}$,... , $z^{(T)}$) sürekli bir gösterime eşlemektir.

Kodlayıcı altı özdeş katmandan oluşan bir yığındır. Altı burada sihirli bir sayı değil, sadece orijinal transformatör makalesinde yapılan hiperparametreli bir seçimdir. Katman sayısını model performansına göre ayarlayabilirsiniz. Bu özdeş katmanların her birinin içinde iki alt katman vardır: biri aşağıda tartışacağımız çok kafalı öz dikkati hesaplar, diğeri ise önceki bölümlerde zaten karşılaştığınız tam bağlantılı bir katmandır.

İlk olarak, bu bölümde daha önce ele alınan ölçeklendirilmiş nokta-çarpım dikkatinin basit bir modifikasyonu olan **çok kafalı öz** dikkatten bahsedelim. Ölçeklendirilmiş nokta çarpımı dikkatinde, girdi dizisini dönüştürmek için üç matris (sorgu, değer ve anahtara karşılık gelen) kullandık. Çok kafalı dikkat bağlamında, bu üç matris kümesini bir dikkat *kafası* olarak düşünebiliriz. Adından da anlaşılacağı üzere, çok kafalı dikkatte, evrişimli sinir ağlarının birden fazla çekirdeğe sahip olabilmesine benzer şekilde, bu tür kafalardan (sorgu, değer ve anahtar matrisleri kümeleri) birden fazlasına sahibiz.

h başlı çok başlı öz dikkat kavramını daha ayrıntılı açıklamak için aşağıdaki adımlara ayıralım.

İlk olarak, $XX\ X\ XXX$, $XX^{(1)(2)}$, … , $XX^{(T)}$) sıralı girdisini okuruz. Her bir elemanın d uzunluğunda bir vektör tarafından gömüldüğünü varsayalım. Burada, girdi bir $T\ Td$ matrisine gömülebilir. Ardından, sorgu, anahtar ve değer öğrenme parametre matrislerinden oluşan h kümeleri oluşturulur:

- U_{qq1} , U_{kk1} , U_{v1}
- U_{qq2} , U_{kk2} , U_{v2}
- …
- $_{qq}\ Uh$, $U_{kk}\ h$, $U\ h_v$

Çünkü bu ağırlık matrislerini her bir xx elemanını$^{(i)}$ gerekli boyut için yansıtmak için kullanıyoruz. matris çarpımlarında sion-matching, hem U_{qq} $_j$ ve U_{kk} $_i$ $d_{kk} \times d$ şeklindedir ve U_v $_j$ var şekli $d_v \times d$. Sonuç olarak, ortaya çıkan her iki dizinin, sorgu ve anahtarın uzunluğu d_{kk} ve ortaya çıkan değer dizisinin uzunluğu d_v olur. Uygulamada, insanlar genellikle basitlik için $d_{kk} = d_v = m$'yi seçerler.

Kodda çok kafalı öz *dikkat* yığınını göstermek için, öncelikle önceki alt bölümde tek sorgu projeksiyon matrisini nasıl oluşturduğumuzu düşünün, *Öz dikkat mekanizmasının parametrelendirilmesi: ölçeklendirilmiş nokta-çarpım dikkati*:

```
>>> torch.manual_seed(123)
>> d = embedded_sentence.shape[1]
>> one_U_query = torch.rand(d, d)
```

Şimdi, orijinal transformatöre benzer sekiz dikkat başlığımız olduğunu, yani $h = 8$ olduğunu varsayalım:

```
>>> h = 8
>> multihead_U_query = torch.rand(h, d, d)
>>> multihead_U_key = torch.rand(h, d, d)
>>> multihead_U_value = torch.rand(h, d, d)
```

Kodda görebileceğimiz gibi, sadece ek bir boyut eklenerek birden fazla dikkat başlığı eklenebilir.

Verileri birden fazla dikkat kafasına bölme

Uygulamada, her bir dikkat kafası için ayrı bir matrise sahip olmak yerine, dönüştürücü uygulamaları tüm dikkat kafaları için tek bir matris kullanır. Dikkat kafaları daha sonra bu matriste Boole maskeleri aracılığıyla erişilebilen mantıksal olarak ayrı bölgeler halinde düzenlenir. Bu, çok kafalı dikkatin daha verimli bir şekilde uygulanmasını mümkün kılar çünkü çoklu matris çarpımları bunun yerine tek bir matris çarpımı olarak uygulanabilir. Ancak, basitlik açısından, bu bölümde bu uygulama detayını atlıyoruz.

İzdüşüm matrislerini başlattıktan sonra, ölçeklendirilmiş nokta çarpımı dikkatinde yapılana benzer şekilde izdüşüm dizilerini hesaplayabiliriz. Şimdi, bir dizi sorgu, anahtar ve değer dizisini hesaplamak yerine, bunların h kümesini hesaplamamız gerekir. Daha resmi olarak, örneğin, j. baştaki ith veri noktası için sorgu projeksiyonunu içeren hesaplama aşağıdaki gibi yazılabilir:

$$qq^{(i)}_{j} = U_{qj}\, xx^{(i)}$$

Daha sonra bu hesaplamayı tüm kafalar $jj\ j\ j1jj\ j\ hj$ için tekrarlarız.

Kodda bu, sorgu olarak ikinci giriş kelimesi için aşağıdaki gibi görünür:

```
>> multihead_query_2 = multihead_U_query.matmul(x_2)
>> multihead_query_2.shape
torch.Size([8, 16])
```

Multihead_query_2 matrisi sekiz satırdan oluşur ve her satır j. dikkat başlığına karşılık gelir.

 alinpikbarbell@gmail.com

Benzer şekilde, her bir kafa için anahtar ve değer dizilerini hesaplayabiliriz:

```
>> multihead_key_2 = multihead_U_key.matmul(x_2)
>> multihead_value_2 = multihead_U_value.matmul(x_2)
>> multihead_key_2[2]
tensor([-1.9619, -0.7701, -0.7280, -1.6840, -1.0801, -1.6778, 0.6763, 0.6547,
        1.4445, -2.7016, -1.1364, -1.1204, -2.4430, -0.5982, -0.8292, -1.4401])
```

Kod çıktısı, üçüncü dikkat başlığı aracılığıyla ikinci giriş elemanının anahtar vektörünü gösterir.

Ancak, anahtar ve değer hesaplamalarını sadece x_2 için değil, tüm girdi dizisi öğeleri için tekrarlamamız gerektiğini unutmayın - buna daha sonra öz dikkati hesaplamak için ihtiyacımız var. Bunu yapmanın basit ve açıklayıcı bir yolu, girdi dizisi katıştırmalarını dikkat kafalarının sayısı olan ilk boyut olarak 8 boyutuna genişletmektir. Bunun için .repeat() yöntemini

```
>> stacked_inputs = embedded_sentence.T.repeat(8, 1, 1)
>>> stacked_inputs.shape
torch.Size([8, 16, 8])
```

kullanıyoruz:

Ardından, torch.bmm() aracılığıyla, tüm anahtarları hesaplamak için dikkat kafaları ile bir toplu matris çarpımı yapabiliriz:

```
>> multihead_keys = torch.bmm(multihead_U_key, stacked_inputs)
>> multihead_keys.shape
torch.Size([8, 16, 8])
```

Bu kodda, artık ilk boyutunda sekiz dikkat başlığını ifade eden bir tensörümüz var. İkinci ve üçüncü boyutlar sırasıyla gömme boyutunu ve kelime sayısını ifade eder. Anahtarların daha sezgisel bir gösterime sahip olması için ikinci ve üçüncü boyutları değiştirelim, yani orijinal girdi dizisi embedded_sentence ile aynı boyutsallığa sahip olsun:

```
>> multihead_keys = multihead_keys.permute(0, 2, 1)
>> multihead_keys.shape
torch.Size([8, 8, 16])
```

Yeniden düzenledikten sonra, ikinci dikkat başlığındaki ikinci anahtar değerine aşağıdaki şekilde erişebiliriz:

```
>> multihead_keys[2, 1]
tensor([-1.9619, -0.7701, -0.7280, -1.6840, -1.0801, -1.6778, 0.6763, 0.6547,
        1.4445, -2.7016, -1.1364, -1.1204, -2.4430, -0.5982, -0.8292, -1.4401])
```

Bunun daha önce multihead_key_2[2] aracılığıyla elde ettiğimiz anahtar değerle aynı olduğunu görebiliriz, bu da karmaşık matris manipülasyonlarımızın ve hesaplamalarımızın doğru olduğunu gösterir. Şimdi bunu değer dizileri için tekrarlayalım:

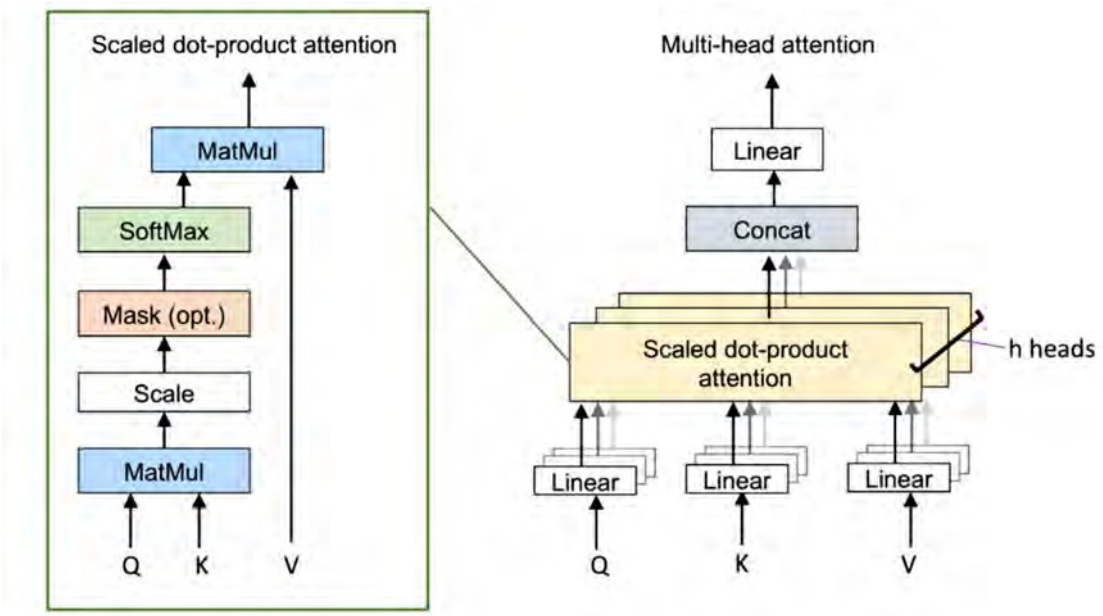
```
>>> multihead_values = torch.matmul(
        multihead_U_value, stacked_inputs)
>> multihead_values = multihead_values.permute(0, 2, 1)
```

Öz-dikkat mekanizmasının parametrelendirilmesi: ölçeklendirilmiş nokta-çarpım dikkati bölümünde açıklandığı gibi bağlam vektörlerini hesaplamak için tek başlı dikkat hesaplama adımlarını takip ediyoruz. Kısalık için ara adımları atlayacağız ve sorgu olarak ikinci girdi öğesi ve rastgele veriler aracılığıyla çoklu kafa_z_2 olarak temsil ettiğimiz sekiz farklı dikkat kafası için bağlam vektörlerini hesapladığımızı varsayacağız:

```
>> multihead_z_2 = torch.rand(8, 16)
```

İlk boyutun sekiz dikkat başlığını indekslediğini ve girdi cümlelerine benzer şekilde bağlam vektörlerinin 16 boyutlu vektörler olduğunu unutmayın. Bu karmaşık görünüyorsa, multihead_z_2'yi *Şekil 16.5*'te gösterilen $z^{(2)}$ 'un sekiz kopyası olarak düşünün; yani, sekiz dikkat başlığının her biri için bir $z^{(2)}$ 'umuz var.

Daha sonra, bu vektörleri $d_v \times h$ uzunluğunda tek bir uzun vektörde birleştiririz ve bunu d_v uzunluğunda bir vektöre geri eşlemek için doğrusal bir projeksiyon (tam bağlı bir katman aracılığıyla) kullanırız. Bu süreç *Şekil 16.7*'de gösterilmektedir:



Şekil 16.7: Ölçeklendirilmiş nokta çarpımı dikkat vektörlerinin tek bir vektörde birleştirilmesi ve doğrusal bir projeksiyondan geçirilmesi

Kodda, birleştirme ve ezme işlemlerini aşağıdaki gibi uygulayabiliriz:

```
>> lineer = torch.nn.Lineer(8*16, 16)
>>  context_vector_2 = linear(multihead_z_2.flatten())
>> context_vector_2.shape
torch.Size([16])
```

 alinpikbarbell@gmail.com

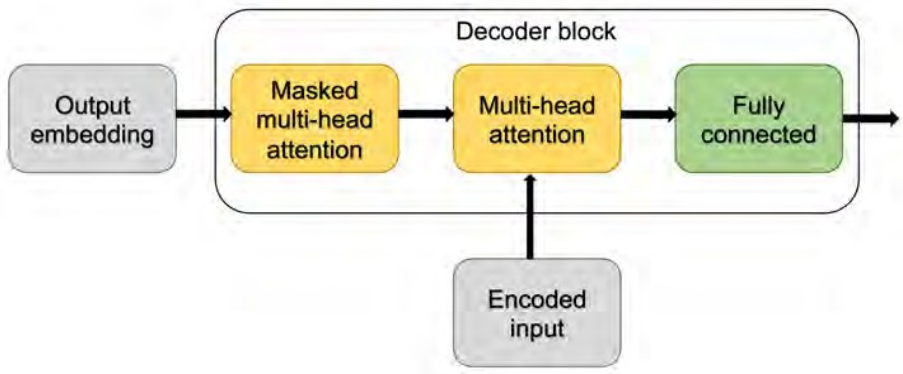
Özetlemek gerekirse, çok kafalı öz dikkat, ölçeklendirilmiş nokta çarpımı dikkat hesaplamasının paralel olarak birden çok kez tekrarlanması ve sonuçların birleştirilmesidir. Pratikte çok iyi çalışır çünkü çoklu kafalar modelin girdinin farklı kısımlarından bilgi yakalamasına yardımcı olur, bu da çoklu çekirdeklerin her kanalın farklı özellik bilgilerini yakalayabildiği bir konvolüsyonel ağda çoklu kanallar üretmesine çok benzer. Son olarak, çok kafalı dikkat hesaplama açısından pahalı gibi görünse de, çoklu kafalar arasında bağımlılık olmadığı için hesaplamanın tamamının paralel olarak yapılabileceğini unutmayın.

Bir dil modeli öğrenme: kod çözücü ve maskeli çoklu kafa dikkati

Kodlayıcıya benzer şekilde, kod **çözücü de** birkaç tekrarlanan katman içerir. Önceki kodlayıcı bölümünde tanıttığımız iki alt katmanın (çok kafalı kendi kendine dikkat katmanı ve tam bağlantılı katman) yanı sıra, her tekrarlanan katman ayrıca maskelenmiş bir çok kafalı dikkat alt katmanı içerir.

Maskelenmiş dikkat, orijinal dikkat mekanizmasının bir varyasyonudur; maskelenmiş dikkat, belirli sayıda kelimeyi "maskeleyerek" modele yalnızca sınırlı bir girdi dizisi geçirir. Örneğin, etiketli bir veri kümesiyle bir dil çeviri modeli oluşturuyorsak, eğitim prosedürü sırasında *i* sırası konumunda, yalnızca 1,...,*i-1* konumlarından doğru çıktı sözcüklerini besleriz. Diğer tüm kelimeler (örneğin, mevcut konumdan sonra gelenler) modelin "hile yapmasını" önlemek için modelden gizlenir. Bu aynı zamanda metin üretiminin doğasıyla da tutarlıdır: gerçek çevrilmiş kelimeler eğitim sırasında bilinmesine rağmen, pratikte temel gerçek hakkında hiçbir şey bilmiyoruz. Bu nedenle, modeli yalnızca *i* konumunda daha önce ürettiği çözümlerle besleyebiliriz.

Şekil 16.8, kod çözücü bloğunda katmanların nasıl düzenlendiğini göstermektedir:



Şekil 16.8: Kod çözücü kısmındaki katman düzeni

İlk olarak, önceki çıktı kelimeleri (çıktı gömüleri) maskelenmiş çok kafalı dikkat katmanına aktarılır. Ardından, ikinci katman hem kodlayıcı bloğundan gelen kodlanmış girdileri hem de maskelenmiş çok kafalı dikkat katmanının çıktısını çok kafalı dikkat katmanına alır. Son olarak, çok kafalı dikkat çıktılarını, genel model çıktısını üreten tam bağlantılı bir katmana aktarıyoruz: çıktı sözcüklerine karşılık gelen bir olasılık vektörü.

Bu kelime olasılıklarından tahmin edilen kelimeleri elde etmek için *Bölüm 15*, Tekrarlayan Sinir *Ağları Kullanarak Sıralı Verileri Modelleme'de tekrarlayan* sinir ağında kullandığımız genel yaklaşıma benzer bir argmax fonksiyonu kullanabileceğimizi unutmayın.

Kod çözücü ile kodlayıcı bloğu karşılaştırıldığında, temel fark, modelin katılabileceği dizi öğeleri aralığıdır. Kodlayıcıda, verilen her bir kelime için dikkat, cümledeki tüm kelimeler üzerinden hesaplanır; bu da bir tür çift yönlü girdi ayrıştırması olarak düşünülebilir. Kod çözücü de kodlayıcıdan çift yönlü ayrıştırılmış girdileri alır. Bununla birlikte, çıkış dizisi söz konusu olduğunda, kod çözücü yalnızca mevcut giriş konumundan önceki öğeleri dikkate alır, bu da tek yönlü giriş ayrıştırma biçimi olarak yorumlanabilir.

Uygulama ayrıntıları: konumsal kodlamalar ve katman normalizasyonu

Bu alt bölümde, şu ana kadar üzerinden geçtiğimiz ancak bahsetmeye değer bazı transformatör uygulama detaylarını tartışacağız.

İlk olarak, *Şekil 16.6'd*aki orijinal dönüştürücü mimarisinin bir parçası olan konumsal **kodlamaları** ele alalım. Konumsal kodlamalar girdi dizisi sıralaması hakkında bilgi yakalamaya yardımcı olur ve dönüştürücülerin önemli bir parçasıdır çünkü hem ölçeklendirilmiş nokta çarpımı dikkat katmanları hem de tam bağlantılı katmanlar permütasyondan bağımsızdır. Bu, konumsal kodlama olmadan kelimelerin sırasının göz ardı edildiği ve dikkat tabanlı kodlamalar için herhangi bir fark yaratmadığı anlamına gelir. Ancak, kelime sırasının bir cümleyi anlamak için gerekli olduğunu biliyoruz. Örneğin, aşağıdaki iki cümleyi ele alalım:

1. Mary John'a bir çiçek verir
2. Yuhanna Meryem'e bir çiçek verir

İki cümlede geçen kelimeler tamamen aynıdır; ancak anlamları çok farklıdır.

Dönüştürücüler, kodlayıcı ve kod çözücü bloklarının başındaki giriş gömülerine küçük değerlerden oluşan bir vektör ekleyerek farklı konumlardaki aynı kelimelerin biraz farklı kodlamalara sahip olmasını sağlar. Özellikle, orijinal dönüştürücü mimarisi sinüzoidal kodlama olarak adlandırılan bir kodlama kullanır:

$$_{(PEii2ii)} = \sin(pos/10000^{2ii/d \text{ modeli}})$$

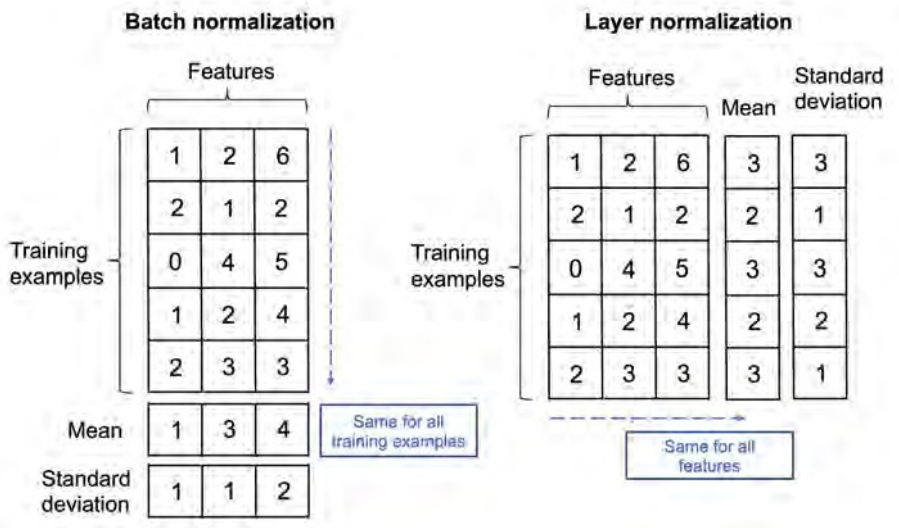
$$_{(PEii2ii1)} = \cos(pos/10000^{2ii/d \text{ modeli}})$$

Burada i sözcüğün konumunu, k ise kodlama vektörünün uzunluğunu göstermektedir. k'yı girdi sözcük gömüleriyle aynı boyuta sahip olacak şekilde seçiyoruz, böylece konumsal kodlama ve sözcük gömüleri birbirine eklenebiliyor. Sinüzoidal fonksiyonlar, konumsal kodlamaların çok büyük olmasını önlemek için kullanılır. Örneğin, mutlak pozisyon 1,2,3..., *n'yi konumsal* kodlamalar olarak kullansaydık, bunlar kelime kodlamasını domine edecek ve kelime gömme değerlerini ihmal edilebilir hale getirecekti.

alinpikbarbell@gmail.com

Genel olarak, *mutlak* (önceki formülde gösterildiği gibi) ve *göreceli olmak* üzere iki tür konumsal kodlama vardır. İlki kelimelerin mutlak konumlarını kaydeder ve bir cümledeki kelime kaymalarına karşı duyarlıdır. Diğer bir deyişle, mutlak konumsal kodlamalar verilen her konum için sabit vektörlerdir. Öte yandan, göreceli kodlamalar yalnızca kelimelerin göreceli konumunu korur ve cümle kaymasına karşı değişmezdir.

Daha sonra, ilk kez 2016 yılında J. Ba, J.R. Kiros ve G.E. Hinton tarafından aynı isimli *Layer Normalization* makalesinde tanıtılan **katman normalleştirme** mekanizmasına bakalım (URL: https://arxiv.org/ abs/1607.06450). *Bölüm 17*, *Yeni Verileri Sentezlemek için Gen- erative Adversarial Networks'te* daha ayrıntılı olarak tartışacağımız toplu normalizasyon, bilgisayarla görme bağlamlarında popüler bir seçim olsa da, katman normalizasyonu, cümle uzunluklarının değişebildiği NLP bağlamlarında tercih edilen bir seçimdir. *Şekil 16.9'da* katman ve toplu normalleştirmenin temel farklılıkları yan yana gösterilmektedir:



Şekil 16.9: Toplu ve katman normalleştirmenin karşılaştırılması

Katman normalizasyonu geleneksel olarak her özellik için bağımsız olarak belirli bir özellikteki tüm öğeler arasında gerçekleştirilirken, dönüştürücülerde kullanılan katman normalizasyonu bu kavramı genişletir ve normalizasyon istatistiklerini her eğitim örneği için bağımsız olarak tüm özellik değerleri arasında birleştirir.

Katman normalizasyonu her eğitim örneği için ortalama ve standart sapmayı hesapladığından, mini parti boyutu kısıtlamalarını veya bağımlılıklarını gevşetir. Yığın normalleştirmenin aksine, katman normalleştirme bu nedenle küçük mini yığın boyutlarına ve değişken uzunluklara sahip verilerden öğrenme yeteneğine sahiptir. Ancak, orijinal dönüştürücü mimarisinin değişken uzunlukta girdilere sahip olmadığını (cümleler gerektiğinde doldurulur) ve RNN'lerin aksine modelde yineleme olmadığını unutmayın. Öyleyse, katman normalleştirmesinin toplu normalleştirme yerine kullanılmasını nasıl gerekçelendirebiliriz? Dönüştürücüler genellikle paralel hesaplama gerektiren çok büyük metin derlemleri üzerinde eğitilir; eğitim örnekleri arasında bağımlılık olan toplu normalleştirme ile bunu başarmak zor olabilir. Katman normalleştirmenin böyle bir bağımlılığı yoktur ve bu nedenle dönüştürücüler için daha doğal bir seçimdir.

Etiketsiz verilerden yararlanarak büyük ölçekli dil modelleri oluşturma

Bu bölümde, orijinal dönüştürücüden ortaya çıkan popüler büyük ölçekli dönüştürücü modellerini tartışacağız. Bu dönüştürücüler arasındaki ortak tema, çok büyük, etiketsiz veri kümeleri üzerinde önceden eğitilmeleri ve daha sonra ilgili hedef görevleri için ince ayar yapılmasıdır. İlk olarak, dönüştürücü tabanlı modellerin ortak eğitim prosedürünü tanıtacağız ve orijinal dönüştürücüden nasıl farklı olduğunu açıklayacağız. Ardından, **Generative Pre-trained Transformer** (**GPT**), **Bidirectional Encoder Representations from Transformers** (**BERT)** ve **Bidirectional and Auto-Regressive Transformers** (**BART**) gibi popüler büyük ölçekli dil modellerine odaklanacağız.

Transformatör modellerinin ön eğitimi ve ince ayarı

Daha önceki bir bölümde, *Dikkat ihtiyacımız olan tek şey: orijinal dönüştürücü mimarisini* tanıtarak, orijinal dönüştürücü mimarisinin dil çevirisi için nasıl kullanılabileceğini tartıştık. Dil çevirisi denetimli bir görevdir ve elde edilmesi çok pahalı olabilen etiketli bir veri kümesi gerektirir. Büyük, etiketli veri kümelerinin eksikliği derin öğrenmede, özellikle de diğer derin öğrenme mimarilerinden daha fazla veriye aç olan transformatör gibi modeller için uzun süredir devam eden bir sorundur. Bununla birlikte, her gün büyük miktarda metin (kitaplar, web siteleri ve sosyal medya gönderileri) üretildiği göz önüne alındığında, model eğitimini iyileştirmek için bu tür etiketsiz verileri nasıl kullanabileceğimiz ilginç bir sorudur.

Dönüştürücülerde etiketsiz verilerden yararlanıp yararlanamayacağımızın cevabı *evettir* ve işin püf noktası **kendi kendine** denetimli **öğrenme** adı verilen bir süreçtir: denetimli öğrenmeden "etiketleri" düz metnin kendisinden üretebiliriz. Örneğin, büyük, etiketsiz bir metin külliyatı verildiğinde, modeli **sonraki kelime** tahminini gerçekleştirecek şekilde eğitiriz, bu da modelin kelimelerin olasılık dağılımını öğrenmesini sağlar ve güçlü bir dil modeli olmak için güçlü bir temel oluşturabilir.

Kendi kendine denetimli öğrenme geleneksel olarak **denetimsiz ön eğitim** olarak da adlandırılır ve modern dönüştürücü tabanlı modellerin başarısı için gereklidir. Denetimsiz ön eğitimdeki "denetimsiz", etiketsiz veri kullandığımız gerçeğini ifade eder; ancak etiketleri oluşturmak için verilerin yapısını kullandığımızdan (örneğin, daha önce bahsedilen sonraki kelime tahmin görevi), bu hala denetimli bir öğrenme sürecidir.

Denetimsiz ön eğitim ve sonraki kelime tahmininin nasıl çalıştığını biraz daha detaylandırmak gerekirse, *n* kelime içeren bir cümlemiz varsa, ön eğitim prosedürü aşağıdaki üç adıma ayrıştırılabilir:

1. Zaman *adımı 1'*de, temel gerçek kelimeleri 1, ..., i-1 ile besleyin.
2. Modelden *i* konumundaki kelimeyi tahmin etmesini ve bunu temel gerçek *i* kelimesi ile karşılaştırmasını isteyin.
3. Modeli ve zaman adımını güncelleyin, $i := i+1$. Adım 1'e geri dönün ve tüm kelimeler elde edilene kadar tekrarlayın.

Bir sonraki yinelemede, modelin bir önceki turda ürettiği kelimeler yerine her zaman modeli temel gerçek (doğru) kelimelerle beslediğimizi belirtmeliyiz.

alinpikbarbell@gmail.com

Ön eğitimin ana fikri, düz metinden yararlanmak ve daha sonra (daha küçük) etiketli bir veri kümesinin mevcut olduğu bazı özel görevleri gerçekleştirmek için modeli aktarmak ve ince ayar yapmaktır. Şimdi, birçok farklı türde ön eğitim tekniği vardır. Örneğin, daha önce bahsedilen sonraki kelime tahmin görevi tek yönlü bir ön eğitim yaklaşımı olarak düşünülebilir. Daha sonra, çeşitli işlevler elde etmek için farklı dil modellerinde kullanılan ek ön eğitim tekniklerini tanıtacağız.

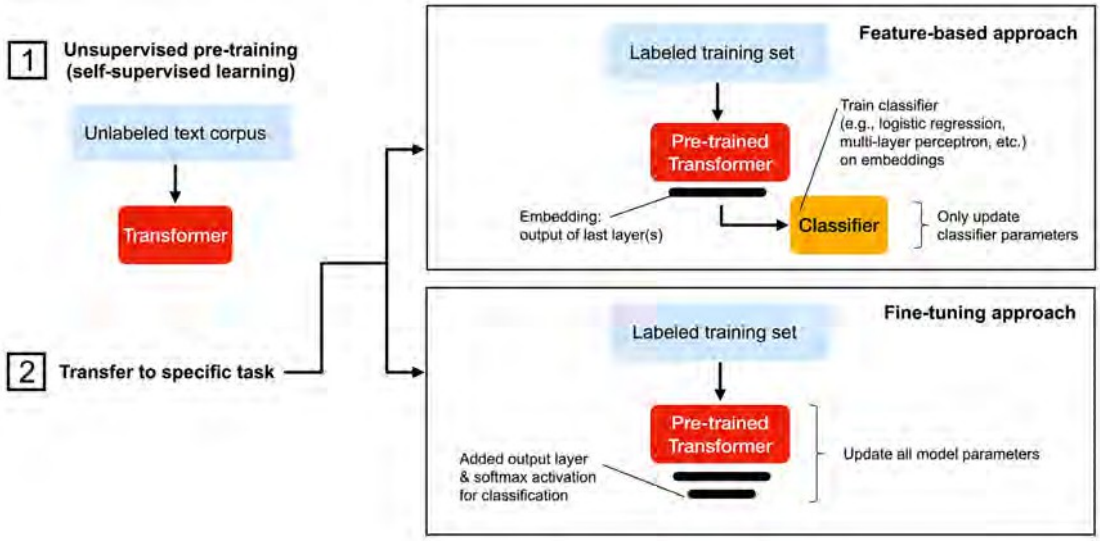
Dönüştürücü tabanlı bir modelin eksiksiz bir eğitim prosedürü iki bölümden oluşur: (1) büyük, etiketsiz bir veri kümesi üzerinde ön eğitim ve (2) etiketli bir veri kümesi kullanarak belirli alt görevler için modeli eğitme (yani, ince ayar yapma). İlk adımda, önceden eğitilmiş model herhangi bir özel görev için tasarlanmaz, bunun yerine "genel" bir dil modeli olarak eğitilir. Daha sonra, ikinci adımda, etiketli bir veri kümesi üzerinde düzenli denetimli öğrenme yoluyla herhangi bir özelleştirilmiş görev için genelleştirilebilir.

Önceden eğitilmiş modelden elde edilebilecek temsillerle, bir modeli belirli bir göreve aktarmak ve uyarlamak için temel olarak iki strateji vardır: (1) **özellik tabanlı** bir **yaklaşım** ve (2) **ince ayar yaklaşımı**. (Burada, bu temsilleri bir modelin son katmanlarının gizli katman aktivasyonları olarak düşünebiliriz).

Özellik tabanlı yaklaşım, önceden eğitilmiş gösterimleri etiketli bir veri kümesine ek özellikler olarak kullanır. Bu, önceden eğitilmiş modelden cümle özelliklerinin nasıl çıkarılacağını öğrenmemizi gerektirir. Bu özellik çıkarma yaklaşımı için iyi bilinen erken bir model, Peters ve meslektaşları tarafından 2018 yılında *Deep Contextualized Word Representa- tions* (URL: https://arxiv.org/abs/1802.05365) makalesinde önerilen **ELMo'**dur (**Embeddings from Language Models**). ELMo, kelimeleri belirli bir oranda maskeleyen önceden eğitilmiş çift yönlü bir dil modelidir. Özellikle, ön eğitim sırasında girdi kelimelerinin yüzde 15'ini rastgele maskeler ve modelleme görevi bu boşlukları doldurmak, yani eksik (maskelenmiş) kelimeleri tahmin etmektir. Bu, daha önce tanıttığımız ve *i*. zaman adımında gelecekteki tüm kelimeleri gizleyen tek yönlü yaklaşımdan farklıdır. Çift yönlü maskeleme, bir modelin her iki uçtan da öğrenmesini sağlar ve böylece bir cümle hakkında daha bütünsel bilgi yakalayabilir. Önceden eğitilmiş ELMo modeli, daha sonra belirli görevler için girdi özellikleri olarak hizmet eden yüksek kaliteli cümle temsilleri oluşturabilir. Başka bir deyişle, özellik tabanlı yaklaşımı, *Bölüm 5, Boyut Azaltma Yoluyla Verileri Sıkıştırma*'da ele aldığımız temel bileşen analizine benzer bir model tabanlı özellik çıkarma tekniği olarak düşünebiliriz.

Öte yandan ince ayar yaklaşımı, önceden eğitilmiş model parametrelerini geri yayılım yoluyla düzenli denetimli bir şekilde günceller. Özellik tabanlı yöntemden farklı olarak, sınıflandırma gibi belirli görevleri yerine getirmek için genellikle önceden eğitilmiş modele başka bir tam bağlı katman ekleriz ve ardından etiketli eğitim setindeki tahmin performansına göre tüm modeli güncelleriz. Bu yaklaşımı izleyen popüler modellerden biri, çift yönlü bir dil modeli olarak önceden eğitilmiş büyük ölçekli bir dönüştürücü modeli olan BERT'tir. BERT'i aşağıdaki alt bölümlerde daha ayrıntılı olarak tartışacağız. Ayrıca, bu bölümün son kısmında, *Bölüm 8, Makine Öğrenimini Duygu Analizine Uygulama* ve *Bölüm 15, Tekrarlayan Sinir Ağları Kullanarak Sıralı Verileri Modelleme*'de çalıştığımız film incelemesi veri kümesini kullanarak duygu sınıflandırması için önceden eğitilmiş bir BERT modeline nasıl ince ayar yapılacağını gösteren bir kod örneği göreceğiz.

Bir sonraki bölüme geçmeden ve popüler dönüştürücü tabanlı dil modellerini tartışmaya başlamadan önce, aşağıdaki şekil dönüştürücü modellerinin eğitiminin iki aşamasını özetler ve özellik tabanlı ve ince ayar yaklaşımları arasındaki farkı gösterir:



Şekil 16.10: Aşağı akış görevleri için önceden eğitilmiş bir dönüştürücüyü benimsemenin iki ana yolu

GPT ile etiketlenmemiş verilerden yararlanma

Generative Pre-trained Transformer (**GPT**), OpenAI tarafından geliştirilen metin üretmeye yönelik popüler bir büyük ölçekli dil modelleri serisidir. Mayıs 2020'de piyasaya sürülen en yeni model olan GPT-3 (*Dil Modelleri Az Atışlı Öğrenicilerdir*) şaşırtıcı sonuçlar üretmektedir. GPT-3 tarafından üretilen metnin kalitesini insan tarafından üretilen metinlerden ayırt etmek çok zordur. Bu bölümde, GPT modelinin yüksek düzeyde nasıl çalıştığını ve yıllar içinde nasıl geliştiğini tartışacağız.

Tablo 16.1'de listelendiği gibi, GPT model serisindeki belirgin bir gelişme parametre sayısıdır:

Model	Yayın yılı	Parametre sayısı	Başlık	Kağıt bağlantısı
GPT-1	2018	110 milyon	Üretken Ön Eğitim ile Dil Anlayışının Geliştirilmesi	https://www.cs.ubc.ca/~amuham01/LING530/papers/radford2018improving.pdf
GPT-2	2019	1,5 milyar	Dil Modelleri Denetimsiz Çok Görevli Öğrenicilerdir	https://www.semanticscholar.org/paper/Language-Models-are-Unsupervised-Multitask-Learners-Radford-Wu/9405cc0d6169988371b2755e573cc28650d14dfe
GPT-3	2020	175 milyar	Dil Modelleri Az Atış Yapan Öğrenicilerdir	https://arxiv.org/pdf/2005.14165.pdf

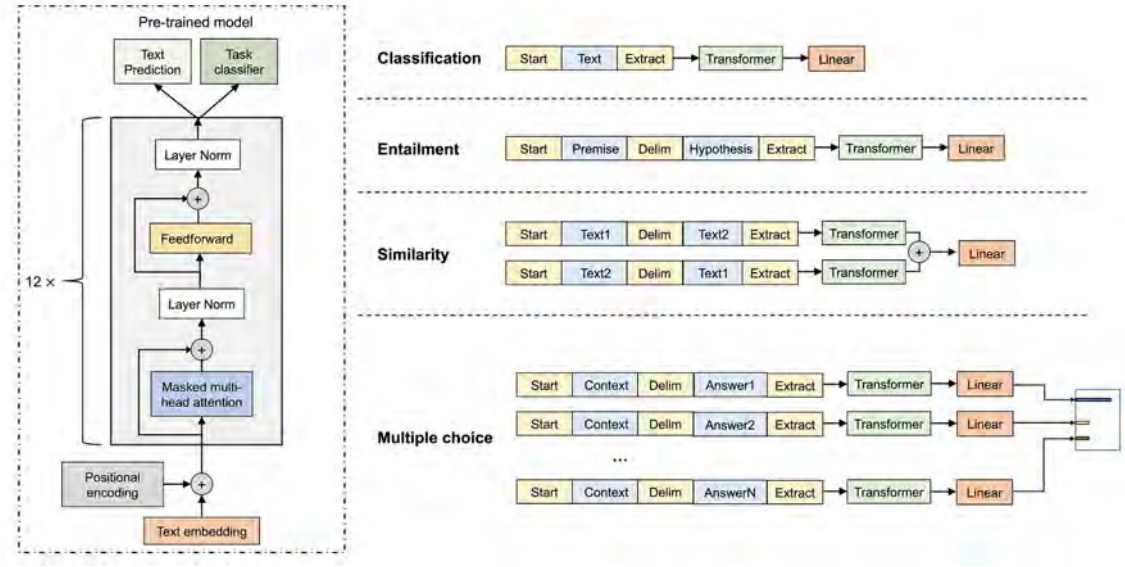
alinpikbarbell@gmail.com

Tablo 16.1: GPT modellerine genel bakış

Ancak haddimizi aşmayalım ve önce 2018'de piyasaya sürülen GPT-1 modeline daha yakından bakalım. Eğitim prosedürü iki aşamaya ayrılabilir:

1. Büyük miktarda etiketlenmemiş düz metin üzerinde ön eğitim
2. Denetimli ince ayar

*Şekil 16.11'de (*GPT-1 makalesinden uyarlanmıştır) gösterildiği gibi, GPT-1'i (1) bir kod çözücü (ve bir kodlayıcı bloğu olmadan) ve (2) belirli görevleri yerine getirmek için denetimli ince ayar için daha sonra eklenen ek bir katmandan oluşan bir dönüştürücü olarak düşünebiliriz:



Şekil 16.11: GPT-1 transformatörü

Şekilde, görevimiz *Metin Tahmini* (bir sonraki kelimeyi tahmin etme) ise, modelin ön eğitim adımından sonra hazır olduğuna dikkat edin. Aksi takdirde, örneğin, görevimiz sınıflandırma veya regresyon ile ilgiliyse, denetimli ince ayar gereklidir.

Ön eğitim sırasında GPT-1, belirli bir kelime konumunda modelin bir sonraki kelimeyi tahmin etmek için yalnızca önceki kelimelere dayandığı bir dönüştürücü kod çözücü yapısı kullanır. GPT-1, BERT'teki gibi çift yönlü bir mekanizmanın aksine (bu bölümde daha sonra ele alacağız) tek yönlü bir öz dikkat mekanizması kullanır, çünkü GPT-1 sınıflandırmadan ziyade metin üretimine odaklanmıştır. Metin oluşturma sırasında, doğal bir soldan sağa yönelimle kelimeleri tek tek üretir. Burada vurgulanması gereken başka bir husus daha vardır: eğitim prosedürü sırasında, her pozisyon için, her zaman önceki pozisyonlardan doğru kelimeleri modele besleriz. Ancak, çıkarım sırasında, yeni metinler üretebilmek için modele sadece ürettiği kelimeleri besliyoruz.

 alinpikbarbell@gmail.com

Önceden eğitilmiş modeli (önceki şekilde *Transformer* olarak etiketlenen blok) elde ettikten sonra, bunu girdi ön işleme bloğu ile doğrusal katmanın bir çıktı katmanı olarak hizmet ettiği doğrusal bir katman arasına yerleştiririz (bu kitapta daha önce tartıştığımız önceki derin sinir ağı modellerine benzer). Sınıflandırma görevleri için ince ayar yapmak, girdiyi önce tokenize etmek ve ardından önceden eğitilmiş modele ve yeni eklenen doğrusal katmana beslemek kadar basittir ve bunu bir softmax aktivasyon fonksiyonu izler. Bununla birlikte, soru cevaplama gibi daha karmaşık görevler için, girdiler önceden eğitilmiş modelle eşleşmesi gerekmeyen belirli bir formatta düzenlenir ve bu da her görev için özelleştirilmiş ekstra bir işlem adımı gerektirir. Belirli değişikliklerle ilgilenen okuyucuların ek ayrıntılar için GPT-1 makalesini okumaları tavsiye edilir (bağlantı önceki tabloda verilmiştir).

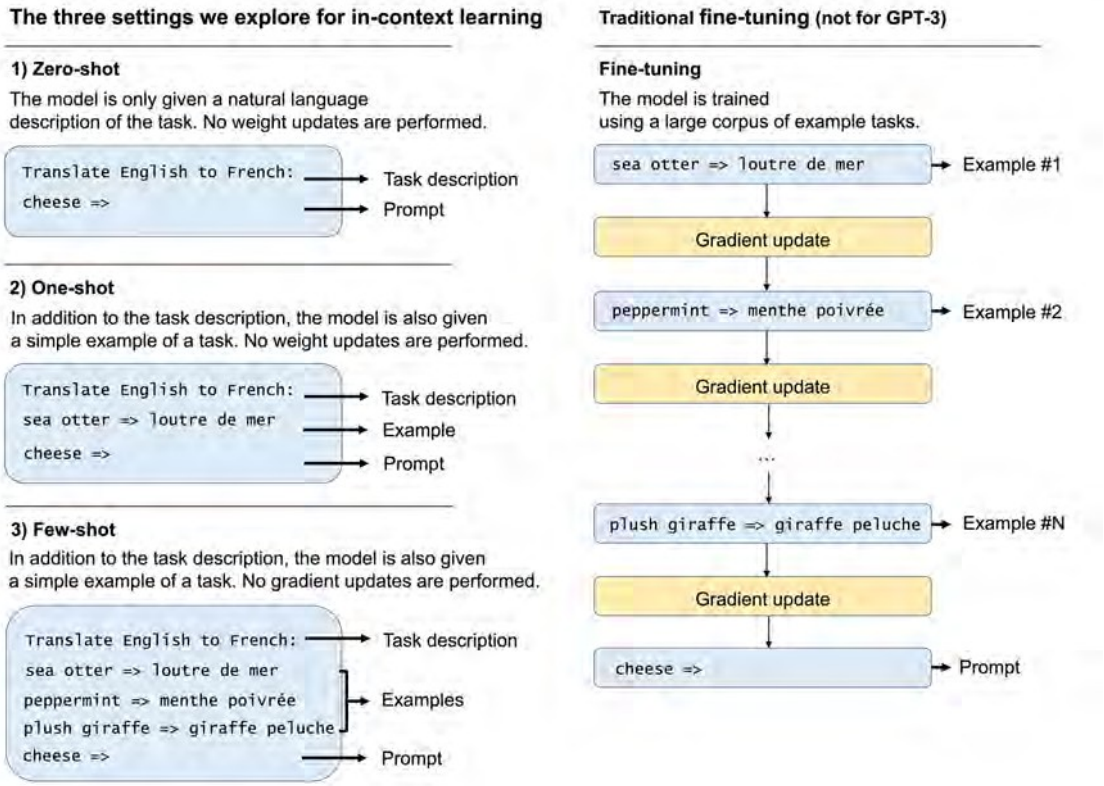
GPT-1 ayrıca **sıfır** atışlı **görevlerde** de şaşırtıcı derecede iyi performans göstererek, göreve özgü minimum ince ayar ile farklı görev türleri için özelleştirilebilen genel bir dil modeli olma yeteneğini kanıtlamaktadır. Sıfır atışlı öğrenme genellikle makine öğreniminde test ve çıkarım sırasında modelin eğitim sırasında gözlemlenmeyen sınıflardan örnekleri sınıflandırması gereken özel bir durumu tanımlar. GPT bağlamında, sıfır atış ayarı görünmeyen görevleri ifade eder.

GPT'nin uyarlanabilirliği araştırmacılara göreve özgü girdi ve model kurulumundan kurtulmaları için ilham vermiş ve bu da GPT-2'nin geliştirilmesine yol açmıştır. Selefinin aksine, GPT-2 artık girdi veya ince ayar aşamalarında herhangi bir ek modifikasyon gerektirmiyor. GPT-2, gerekli formata uyacak şekilde dizileri yeniden düzenlemek yerine, farklı girdi türlerini ayırt edebilir ve "bağlamlar" olarak adlandırılan küçük ipuçlarıyla ilgili aşağı akış görevlerini gerçekleştirebilir. Bu, yalnızca girdiye koşullandırmak yerine, hem girdi hem de görev türüne koşullandırılmış çıktı olasılıklarının ($p(output|input, task)$) modellenmesiyle elde edilir. Örneğin, bağlam Fransızca'ya çevir, İngilizce metin, Fransızca metin içeriyorsa modelin bir çeviri görevini tanıması beklenir.

Bu kulağa GPT'den çok daha "yapay olarak akıllı" geliyor ve gerçekten de model boyutunun yanı sıra en dikkat çekici gelişme. İlgili makalenin başlığının da belirttiği gibi (D*il Modelleri Denetimsiz Çok Görevli Öğrenicilerdir)*, denetimsiz bir dil modeli sıfır atışlı öğrenmenin anahtarı olabilir ve GPT-2 bu çok görevli öğreniciyi oluşturmak için sıfır atışlı görev aktarımından tam olarak yararlanır.

GPT-2 ile karşılaştırıldığında GPT-3, odağı bağlam içi öğrenme yoluyla sıfırdan tek atımlı ve **birkaç atımlı öğrenmeye** kaydırması anlamında daha az "iddialıdır". Göreve özgü hiçbir eğitim örneği sağlamamak çok katı gibi görünse de, birkaç atışlık öğrenme sadece daha gerçekçi değil, aynı zamanda daha insani bir yöntemdir: insanlar yeni bir görevi öğrenebilmek için genellikle birkaç örnek görmeye ihtiyaç duyarlar. Adından da anlaşılacağı gibi, tek seferlik öğrenme tam olarak bir örnekl e sınırlıyken, birkaç seferlik öğrenme modelin görevin *birkaç örneğini* gördüğü anlamına gelir.

Şekil 16.12'de sıfır atış, tek atış, birkaç atış ve ince ayar işlemleri arasındaki fark gösterilmektedir:



Şekil 16.12: Sıfır atışlı, tek atışlı ve birkaç atışlı öğrenmenin karşılaştırılması

GPT-3'ün model mimarisi, 100 katlık parametre boyutu artışı ve seyrek bir dönüştürücü kullanımı dışında GPT-2 ile hemen hemen aynıdır. Daha önce tartıştığımız orijinal (yoğun) dikkat mekanizmasında, her eleman girdideki diğer tüm elemanlara katılır ve bu da $O(n^2)$ karmaşıklığı ile ölçeklenir. **Seyrek dikkat,** yalnızca normalde $n^{1/p}$ ile orantılı olan sınırlı boyuttaki öğelerin bir alt kümesine katılarak verimliliği artırır. İlgilenen okuyucular sparse transformer makalesini ziyaret ederek spesifik alt küme seçimi hakkında daha fazla bilgi edinebilirler: *Generating Long Sequences with Sparse Transformers by Rewon Child et al*. 2019 (URL: https://arxiv.org/abs/1904.10509).

Yeni metin oluşturmak için GPT-2'yi kullanma

Bir sonraki dönüştürücü mimarisine geçmeden önce, yeni metin oluşturmak için en yeni GPT modellerini nasıl kullanabileceğimize bir göz atalım. GPT-3'ün hala nispeten yeni olduğunu ve şu anda yalnızca https://openai.com/blog/openai-api/ adresindeki OpenAI API aracılığıyla beta sürümü olarak mevcut olduğunu unutmayın. Bununla birlikte, GPT-2'nin bir uygulaması Hugging Face (popüler bir NLP ve makine öğrenimi şirketi; http://huggingface.co) tarafından kullanıma sunulmuştur ve biz de bunu kullanacağız.

 alinpikbarbell@gmail.com

GPT-2'ye, Hugging Face tarafından oluşturulan ve ön eğitim ve ince ayar için çeşitli transformatör tabanlı modeller sağlayan çok kapsamlı bir Python kütüphanesi olan transformatörler aracılığıyla erişeceğiz. Kullanıcılar ayrıca özelleştirilmiş modellerini forumda tartışabilir ve paylaşabilir. İlgileniyorsanız topluluğa göz atmaktan ve etkileşimde bulunmaktan çekinmeyin: https://discuss.huggingface.co.

Transformatörlerin 4.9.1 sürümünü yükleme

Bu paket hızla geliştiğinden, aşağıdaki alt bölümlerdeki sonuçları tekrarlayamayabilirsiniz. Referans olarak, bu eğitimde Haziran 2021'de yayınlanan 4.9.1 sürümü kullanılmıştır. Bu kitapta kullandığımız sürümü yüklemek için terminalinizde aşağıdaki komutu çalıştırarak PyPI'dan yükleyebilirsiniz:

```
pip install transformers==4.9.1
```

Ayrıca resmi kurulum sayfasındaki en son talimatları kontrol etmenizi öneririz:

https://huggingface.co/transformers/installation.html

Transformers kütüphanesini kurduktan sonra, yeni metin oluşturabilen önceden eğitilmiş bir GPT modelini içe aktarmak için aşağıdaki kodu çalıştırabiliriz:

```
>> from transformers import pipeline, set_seed
>> generator = pipeline('text-generation', model='gpt2')
```

Ardından, modele bir metin parçacığı sorabilir ve bu girdi parçacığına dayalı olarak yeni metin oluşturmasını isteyebiliriz:

```
>> set_seed(123)
>> generator("Hey okuyucular, bugün",
...              max_length=20,
...              num_return_sequences=3)

[{'generated_text': "Hey okuyucular, bugün en sevdiğimiz indie rock
gruplarından birini son kez görmeyeceğiz"},
 {'generated_text': 'Hey okuyucular, bugün Noel. Bu N o e l  değil, çünkü Noel çok
uzun ve umarım'},
 {'generated_text': "Hey okuyucular, bugün CTA Günü!\n\nÖzel bir etkinliğe ev
sahipliği yapmaktan gurur duyuyoruz"}]
```

Çıktıdan da görebileceğimiz gibi, model metin parçacığımıza dayanarak üç makul cümle oluşturdu. Daha fazla örnek keşfetmek istiyorsanız, lütfen rastgele tohumu ve maksimum dizi uzunluğunu değiştirmekten çekinmeyin.

Ayrıca, daha önce *Şekil 16.10*'da gösterildiği gibi, diğer modelleri eğitmek üzere özellikler oluşturmak için bir dönüştürücü modeli kullanabiliriz. Aşağıdaki kod, bir girdi metnine dayalı özellikler oluşturmak için GPT-2'yi nasıl kullanabileceğimizi göstermektedir:

```
>> from transformers import GPT2Tokenizer
>>> tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
>> metin = "Bu cümleyi kodlayalım"
>> encoded_input = tokenizer(text, return_tensors='pt')
>>> encoded_input
{'input_ids': tensor([[ 5756,      514, 37773,     428, 6827]]), 'attention_mask':
tensör([[1, 1, 1, 1]])}
```

Bu kod girdi cümle metnini GPT-2 modeli için tokenize edilmiş bir formata kodlamıştır. Gördüğümüz gibi, dizeleri bir tamsayı gösterimine eşledi ve dikkat maskesini tüm 1'lere ayarladı, bu da burada gösterildiği gibi kodlanmış girdiyi modele ilettiğimizde tüm kelimelerin işleneceği anlamına geliyor:

```
>> from transformers import GPT2Model
>>> model = GPT2Model.from_pretrained('gpt2')
>>> çıktı = model(**kodlanmış_girdi)
```

Çıkış değişkeni son gizli durumu, yani giriş cümlesinin GPT-2 tabanlı özellik kodlamasını saklar:

```
>> output['last_hidden_state'].shape
torch.Size([1, 5, 768])
```

Ayrıntılı çıktıyı bastırmak için sadece tensörün şeklini gösterdik. İlk boyutu yığın boyutudur (yalnızca bir girdi metnimiz var), bunu cümle uzunluğu ve özellik kodlamasının boyutu takip eder. Burada, beş kelimenin her biri 768 boyutlu bir vektör olarak kodlanmıştır.

Şimdi, bu özellik kodlamasını belirli bir veri kümesine uygulayabilir ve *Bölüm 8, Makine Öğrenimini Duygu Analizine Uygulama*'da tartışıldığı gibi bir kelime torbası modeli kullanmak yerine GPT-2 tabanlı özellik temsiline dayalı bir aşağı akış sınıflandırıcısını eğitebiliriz.

Ayrıca, önceden eğitilmiş büyük dil modellerini kullanmaya alternatif bir yaklaşım da daha önce tartıştığımız gibi ince ayardır. Bu bölümün ilerleyen kısımlarında bir ince ayar örneği göreceğiz.

GPT-2'nin kullanımıyla ilgili ek ayrıntılarla ilgileniyorsanız, aşağıdaki doküman sayfalarını tavsiye ederiz:

- https://huggingface.co/gpt2
- https://huggingface.co/docs/transformers/model_doc/gpt2

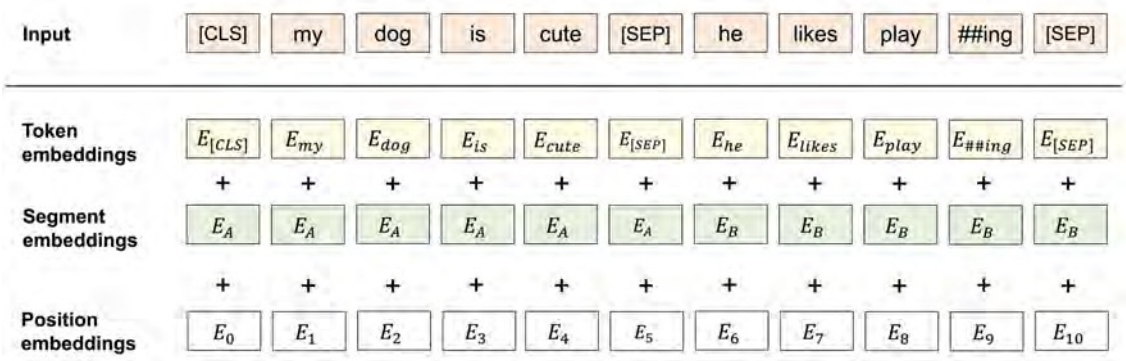
 alinpikbarbell@gmail.com

BERT ile çift yönlü ön eğitim

Tam adı **Bidirectional Encoder Representations from Transformers olan BERT,** 2018 yılında bir Google araştırma ekibi tarafından oluşturulmuştur (BERT*: Pre-training of Deep Bidirectional Transformers for Language Understanding*, *J. Devlin, M. Chang, K. Lee,* and *K. Toutanova,* https://arxiv.org/abs/1810.04805). Referans olarak, farklı mimariler oldukları için GPT ve BERT'i doğrudan karşılaştıramasak da, BERT 345 milyon parametreye sahiptir (bu da onu GPT-1'den sadece biraz daha büyük yapar ve boyutu GPT-2'nin sadece 1/5'i kadardır).

Adından da anlaşılacağı üzere BERT, çift yönlü bir eğitim prosedürü kullanan transformatör-kodlayıcı tabanlı bir model yapısına sahiptir. (Ya da daha doğru bir ifadeyle, BERT'in "yönsüz" eğitim kullandığını düşünebiliriz çünkü tüm girdi öğelerini bir kerede okur). Bu ayar altında, belirli bir kelimenin kodlanması hem önceki hem de sonraki kelimelere bağlıdır. GPT'de girdi öğelerinin doğal bir soldan sağa sırayla okunduğunu ve bunun da güçlü bir üretici dil modeli oluşturmaya yardımcı olduğunu hatırlayın. Çift yönlü eğitim, BERT'in kelime kelime cümle üretme yeteneğini devre dışı bırakır, ancak model artık bilgiyi her iki yönde de işleyebildiğinden, sınıflandırma gibi diğer görevler için daha yüksek kalitede girdi kodlamaları sağlar.

Bir dönüştürücünün kodlayıcısında, belirteç kodlamasının konumsal kodlamaların ve belirteç gömmelerinin bir toplamı olduğunu hatırlayın. BERT kodlayıcısında, bu belirtecin hangi segmente ait olduğunu gösteren ek bir segment gömme vardır. Bu, *Şekil 16.13'te* gösterildiği gibi, her bir belirteç gösteriminin üç bileşen içerdiği anlamına gelir:



Şekil 16.13: BERT kodlayıcı için girişlerin hazırlanması

BERT'te bu ek segment bilgisine neden ihtiyaç duyuyoruz? Bu segment bilgisine duyulan ihtiyaç, BERT'in *sonraki cümle tahmini* adı verilen özel ön eğitim görevinden kaynaklanmaktadır. Bu ön eğitim görevinde, her eğitim örneği iki cümle içerir ve bu nedenle birinci veya ikinci cümleye ait olup olmadığını belirtmek için özel segment notasyonu gerektirir.

6587191CA584E

Şimdi, BERT'in ön eğitim görevlerine daha ayrıntılı olarak bakalım. Diğer tüm dönüştürücü tabanlı dil modellerine benzer şekilde, BERT'in iki eğitim aşaması vardır: ön eğitim ve ince ayar. Ve ön eğitim iki denetimsiz görev içerir: *maskelenmiş dil modelleme* ve *sonraki cümle tahmini*.

Maskeli dil modelinde (**MLM**), belirteçler rastgele olarak *maske belirteçleri* [MASK] olarak adlandırılan belirteçlerle değiştirilir ve modelin bu gizli kelimeleri tahmin etmesi gerekir. GPT'deki sonraki kelime tahminiyle karşılaştırıldığında, BERT'teki MLM daha çok "boşlukları doldurmaya" benzer çünkü model cümledeki tüm belirteçlere (maskelenmiş olanlar hariç) katılabilir. Bununla birlikte, [MASK] belirteçleri normal metinlerde görünmediğinden, kelimeleri basitçe maskelemek ön eğitim ve ince ayar arasında tutarsızlıklara neden olabilir. Bunu hafifletmek için, maskeleme için seçilen kelimelerde daha fazla değişiklik yapılmıştır. Örneğin, BERT'teki kelimelerin yüzde 15'i maskeleme için işaretlenmiştir. Rastgele seçilen bu yüzde 15'lik sözcükler daha sonra aşağıdaki gibi işlenir:

1. Zamanın yüzde 10'unda kelimeyi değiştirmeyin
2. Orijinal sözcük belirtecini zamanın yüzde 10'unda rastgele bir sözcükle değiştirin
3. Orijinal sözcük belirtecini yüzde 80 oranında bir maske belirteci olan [MASK] ile değiştirin

MASK] belirteçlerini eğitim prosedürüne dahil ederken ön eğitim ve ince ayar arasındaki yukarıda bahsedilen tutarsızlıktan kaçınmanın yanı sıra, bu değişikliklerin başka faydaları da vardır. İlk olarak, değiştirilmemiş kelimeler orijinal belirtecin bilgisini koruma olasılığını içerir; aksi takdirde, model yalnızca bağlamdan öğrenebilir ve maskelenmiş kelimelerden hiçbir şey öğrenemez. İkinci olarak, yüzde 10'luk rastgele kelimeler modelin tembelleşmesini, örneğin kendisine verileni geri getirmekten başka bir şey öğrenmemesini önler. Maskeleme, rastgele hale getirme ve kelimeleri değiştirmeden bırakma olasılıkları bir ablasyon çalışmasıyla seçilmiştir (bkz. GPT-2 makalesi); örneğin, yazarlar farklı ayarları test etmiş ve bu kombinasyonun en iyi sonucu verdiğini bulmuşlardır.

Şekil 16.14, tilki kelimesinin maskelendiği ve belirli bir olasılıkla değişmeden kaldığı ya da [MASK] veya *kahve ile değiştirildiği* bir örneği göstermektedir. Modelin daha sonra *Şekil 16.14'te* gösterildiği gibi maskelenmiş (vurgulanmış) kelimenin ne olduğunu tahmin etmesi gerekmektedir:

Şekil 16.14: Bir MLM örneği

Sonraki cümle tahmini, BERT'in çift yönlü kodlaması dikkate alındığında sonraki kelime tahmini görevinin doğal bir modifikasyonudur. Aslında, soru cevaplama gibi birçok önemli NLP görevi, belgedeki iki cümlenin ilişkisine bağlıdır. Bu tür bir ilişkiyi normal dil modelleriyle yakalamak zordur çünkü sonraki kelime tahmini eğitimi, girdi uzunluğu kısıtlamaları nedeniyle genellikle tek bir cümle düzeyinde gerçekleşir.

 alinpikbarbell@gmail.com

Sonraki cümle tahmin görevinde, modele aşağıdaki formatta A ve B olmak üzere iki cümle verilir:

[CLS] A [SEP] B [SEP]

[CLS], kod çözücü çıktısında tahmin edilen etiket için bir yer tutucu görevi gören bir sınıflandırma belirtecinin yanı sıra cümlelerin başlangıcını belirten bir belirteçtir. Öte yandan [SEP] belirteci her cümlenin sonunu belirtmek için eklenmiştir. Modelin daha sonra B'nin A'nın bir sonraki cümlesi ("IsNext") olup olmadığını sınıflandırması gerekmektedir. Modele dengeli bir veri kümesi sağlamak için örneklerin yüzde 50'si "IsNext" olarak etiketlenirken geri kalan örnekler "NotNext" olarak etiketlenir.

BERT, maskelenmiş cümleler ve sonraki cümle tahmini olmak üzere bu iki görev üzerinde aynı anda ön eğitime tabi tutulur. Burada BERT'in eğitim amacı, her iki görevin birleşik kayıp fonksiyonunu en aza indirmektir.

Önceden eğitilmiş modelden başlayarak, ince ayar aşamasında farklı alt görevler için belirli değişiklikler gereklidir. Her girdi örneğinin belirli bir formata uyması gerekir; örneğin, bir [CLS] belirteci ile başlamalı ve birden fazla cümleden oluşuyorsa [SEP] belirteçleri kullanılarak ayrılmalıdır.

Kabaca söylemek gerekirse, BERT dört görev kategorisinde ince ayar yapılabilir: (a) cümle çifti sınıflandırması;
(b) tek cümle sınıflandırması; (c) soru cevaplama; (d) tek cümle etiketleme.

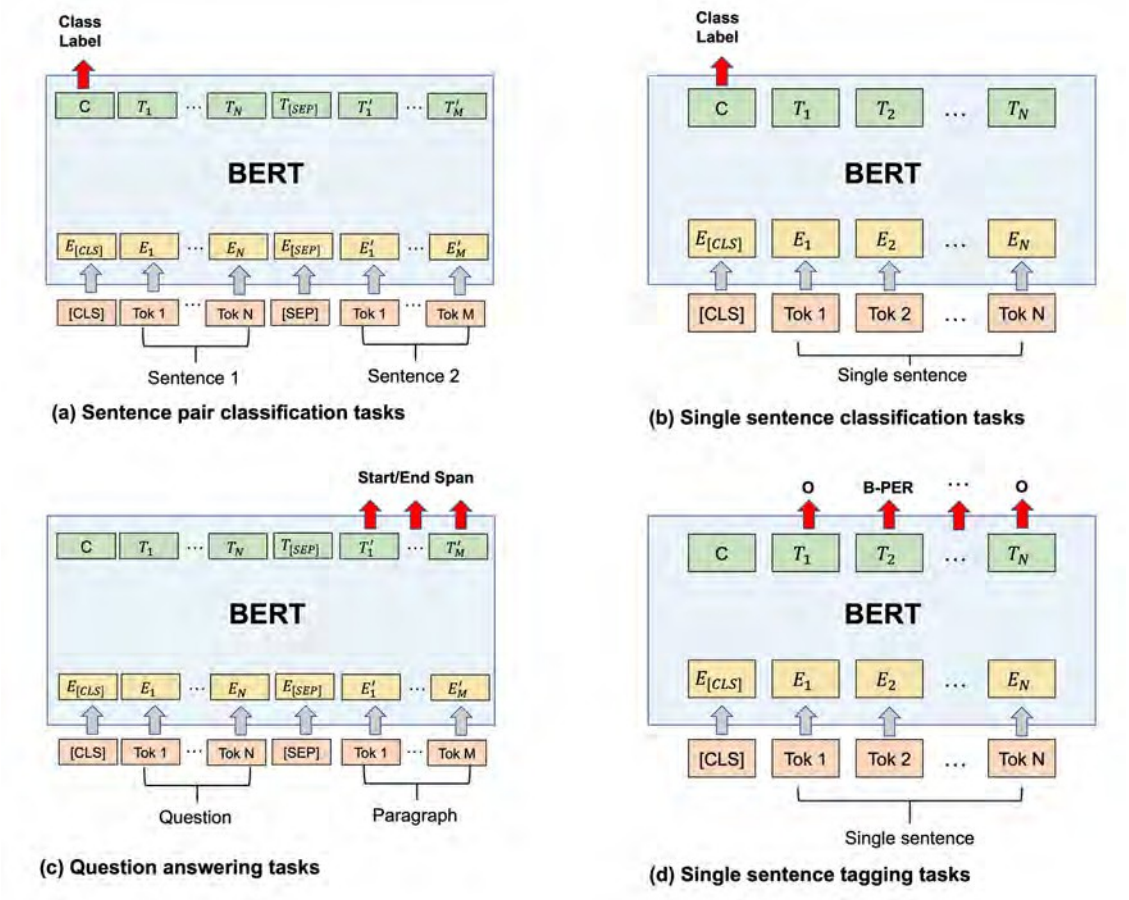
Bunlar arasında (a) ve (b), [CLS] belirtecinin çıktı temsiline yalnızca ek bir softmax katmanı eklenmesini gerektiren sıra düzeyinde sınıflandırma görevleridir. (c) ve (d) ise belirteç düzeyinde sınıflandırma görevleridir. Bu, modelin her bir belirteç için bir sınıf etiketi tahmin etmek üzere ilgili tüm belirteçlerin çıktı temsillerini softmax katmanına aktardığı anlamına gelir.

Soru cevaplama

Görev (c), soru cevaplama, duygu sınıflandırma veya konuşma etiketleme gibi diğer popüler sınıflandırma görevlerine kıyasla daha az tartışılmış görünmektedir. Soru cevaplamada, her girdi örneği soru ve soruyu cevaplamaya yardımcı olan paragraf olmak üzere iki parçaya ayrılabilir. Modelin, soruya uygun bir cevap oluşturan paragraftaki hem başlangıç hem de bitiş belirtecini göstermesi gerekir. Bu, modelin paragraftaki her bir belirteç için bu belirtecin bir başlangıç ya da bitiş belirteci olup olmadığını belirten bir etiket oluşturması gerektiği anlamına gelir. Bir yan not olarak, çıktının başlangıç belirtecinden önce görünen bir bitiş belirteci içerebileceğini ve bunun da yanıt oluşturulurken bir çakışmaya yol açacağını belirtmek gerekir. Bu tür bir çıktı, soruya "Yanıt Yok" olarak tanınacaktır.

6587191CA584E

Şekil 16.15'te gösterildiği gibi, model ince ayar kurulumu çok basit bir yapıya sahiptir: önceden eğitilmiş bir BERT'e bir giriş kodlayıcı eklenir ve sınıflandırma için bir softmax katmanı eklenir. Model yapısı kurulduktan sonra, tüm parametreler öğrenme süreci boyunca ayarlanacaktır.



Şekil 16.15: Farklı dil görevlerinde ince ayar yapmak için BERT kullanımı

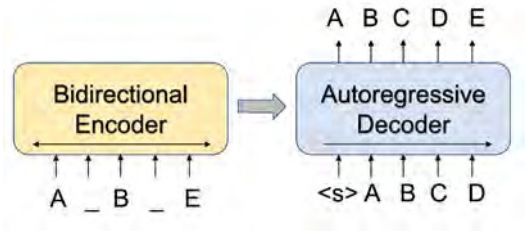
Her iki dünyanın da en iyisi: BART

BART olarak kısaltılan **Bidirectional and Auto-Regressive Transformer**, 2019 yılında Facebook AI Research'teki araştırmacılar tarafından geliştirilmiştir: BART*: Denoising Sequence-to-Sequence Pre-training for Nat- ural Language Generation*, *Translation, and Comprehension, Lewis* ve meslektaşları, https://arxiv.org/ abs/1910.13461. Önceki bölümlerde GPT'nin bir dönüştürücünün kod çözücü yapısını, BERT'in ise bir dönüştürücünün kodlayıcı yapısını kullandığını iddia ettiğimizi hatırlayın. Dolayısıyla bu iki model farklı görevleri iyi bir şekilde yerine getirebilmektedir: GPT'nin uzmanlık alanı metin üretmek iken, BERT sınıflandırma görevlerinde daha iyi performans gösterir. BART, hem GPT hem de BERT'in bir genellemesi olarak görülebilir. Bu bölümün başlığından da anlaşılacağı üzere, BART metin üretme ve sınıflandırma görevlerinin her ikisini de yerine getirebilmektedir. Her iki görevi de iyi bir şekilde yerine getirebilmesinin nedeni, modelin çift yönlü bir kodlayıcının yanı sıra soldan sağa otoregresif bir kod çözücü ile birlikte gelmesidir.

alinpikbarbell@gmail.com

Bunun orijinal transformatörden ne farkı olduğunu merak edebilirsiniz. Aktivasyon fonksiyonu seçimleri gibi bazı küçük değişikliklerin yanı sıra model boyutunda da birkaç değişiklik var. Ancak en ilginç değişikliklerden biri BART'ın farklı model girdileriyle çalışmasıdır. Orijinal dönüştürücü modeli dil çevirisi için tasarlanmıştır, bu nedenle iki girdi vardır: kodlayıcı için çevrilecek metin (kaynak dizisi) ve kod çözücü için çeviri (hedef dizisi). Ayrıca, kod çözücü daha önce *Şekil 16.6'da* gösterildiği gibi kodlanmış kaynak dizisini de alır. Ancak BART'ta girdi formatı, girdi olarak yalnızca kaynak dizisini kullanacak şekilde genelleştirilmiştir. BART, kaybı hesaplamak ve modele ince ayar yapmak için hala bir hedef dizinin gerekli olduğu dil çevirisi de dahil olmak üzere daha geniş bir görev yelpazesini gerçekleştirebilir, ancak bunu doğrudan kod çözücüye beslemek gerekli değildir.

Şimdi BART'ın model yapısına daha yakından bakalım. Daha önce de belirtildiği gibi, BART çift yönlü bir kodlayıcı ve otoregresif bir kod çözücüden oluşmaktadır. Bir eğitim örneği düz metin olarak alındığında, girdi önce "bozulur" ve ardından kodlayıcı tarafından kodlanır. Bu girdi kodlamaları daha sonra üretilen belirteçlerle birlikte kod çözücüye aktarılacaktır. Kodlayıcı çıktısı ve orijinal metin arasındaki çapraz entropi kaybı hesaplanacak ve daha sonra öğrenme süreci ile optimize edilecektir. Kod çözücüye girdi olarak farklı dillerde iki metnin bulunduğu bir dönüştürücü düşünün: çevrilecek ilk metin (kaynak metin) ve hedef dilde oluşturulan metin. BART, birincisini bozuk metinle, ikincisini ise girdi metninin kendisiyle değiştirmek olarak anlaşılabilir.



Şekil 16.16: BART'ın model yapısı

Bozma adımını biraz daha detaylı açıklamak gerekirse, BERT ve GPT'nin maskelenmiş kelimelerin yeniden yapılandırılmasıyla önceden eğitildiğini hatırlayın: BERT "boşlukları doldurur" ve GPT "bir sonraki kelimeyi tahmin eder". Bu ön eğitim görevleri bozuk cümlelerin yeniden yapılandırılması olarak da kabul edilebilir çünkü kelimeleri maskelemek bir cümleyi bozmanın bir yoludur. BART, temiz metne uygulanabilecek aşağıdaki bozulma yöntemlerini sağlar:

- Jeton maskeleme
- Belirteç silme
- Metin doldurma
- Cümle permütasyonu
- Belge rotasyonu

Yukarıda listelenen tekniklerden biri veya daha fazlası aynı cümleye uygulanabilir; en kötü senaryoda, tüm bilgilerin kirlendiği ve bozulduğu durumlarda, metin işe yaramaz hale gelir. Dolayısıyla, kodlayıcının sınırlı bir faydası vardır ve yalnızca kod çözücü modülün düzgün çalışmasıyla model esasen tek yönlü bir dile daha çok benzeyecektir.

6587191CA584E

BART, (a) dizi sınıflandırması dahil olmak üzere çok çeşitli aşağı akış görevlerinde ince ayar yapılabilir, (b) token sınıflandırma, (c) dizi oluşturma ve (d) makine çevirisi. BERT'te olduğu gibi, farklı görevleri yerine getirmek için girdilerde küçük değişiklikler yapılması gerekir.

Sıra sınıflandırma görevinde, BERT'teki [CLS] belirtecine benzer şekilde, oluşturulan etiket belirteci olarak hizmet etmek için girişe ek bir belirteç eklenmesi gerekir. Ayrıca, girişi bozmak yerine, bozulmamış giriş hem kodlayıcıya hem de kod çözücüye beslenir, böylece model girişi tam olarak kullanabilir.

Belirteç sınıflandırması için ek belirteçler gereksiz hale gelir ve model, sınıflandırma için her belirteç için oluşturulan temsili doğrudan kullanabilir.

BART'ta dizi üretimi, kodlayıcının varlığı nedeniyle GPT'den biraz farklıdır. Sıfırdan metin üretmek yerine, BART aracılığıyla dizi üretme görevleri, modele bir bağlamlar külliyatı verildiği ve belirli sorulara bir özet veya soyutlayıcı bir cevap üretmesinin istendiği özetleme ile daha çok karşılaştırılabilir. Bu amaçla, kod çözücü otoregresif olarak çıktı üretirken tüm giriş dizileri kodlayıcıya beslenir.

Son olarak, BART ile orijinal dönüştürücü arasındaki benzerlik göz önüne alındığında BART'ın makine çevirisi yapması doğaldır. Ancak araştırmacılar, orijinal dönüştürücüyü eğitmekle aynı prosedürü izlemek yerine, tüm BART modelini önceden eğitilmiş bir kod çözücü olarak dahil etme olasılığını göz önünde bulundurmuştur. Çeviri modelini tamamlamak için, yeni ve ek bir kodlayıcı olarak rastgele başlatılmış parametrelerden oluşan yeni bir set eklenir. Ardından, ince ayar aşaması iki adımda gerçekleştirilebilir:

1. İlk olarak, kodlayıcı hariç tüm parametreleri dondurun
2. Ardından, modeldeki tüm parametreleri güncelleyin

BART, çeşitli görevler için çeşitli kıyaslama veri kümeleri üzerinde değerlendirilmiş ve BERT gibi diğer ünlü dil modellerine kıyasla çok rekabetçi sonuçlar elde etmiştir. Özellikle, soyut soru yanıtlama, diyalog yanıtı ve özetleme görevleri de dahil olmak üzere üretim görevleri için BART en iyi sonuçları elde etmiştir.

PyTorch'ta bir BERT modeline ince ayar yapma

Orijinal transformatör ve popüler transformatör tabanlı modellerin arkasındaki tüm gerekli kavramları ve teoriyi tanıttığımıza ve tartıştığımıza göre, şimdi daha pratik kısma bir göz atmanın zamanı geldi! Bu bölümde, PyTorch'ta **duygu sınıflandırması** için bir BERT modeline nasıl ince ayar yapacağınızı öğreneceksiniz.

Aralarından seçim yapabileceğiniz birçok transformatör tabanlı model olmasına rağmen, BERT'in model popülerliği ile tek bir GPU'da ince ayar yapılabilmesi için yönetilebilir bir model boyutuna sahip olmak arasında güzel bir denge sağladığını unutmayın. Ayrıca, Hugging Face tarafından sağlanan ve ince ayar için hazır bir grup önceden eğitilmiş model içeren transformers Python paketinin mevcudiyeti göz önüne alındığında, bir BERT'i sıfırdan önceden eğitmenin acı verici ve oldukça gereksiz olduğunu unutmayın.

alinpikbarbell@gmail.com

İlerleyen bölümlerde, IMDb film incelemesi veri kümesinin nasıl hazırlanacağını ve tokenize edileceğini ve duygu sınıflandırması gerçekleştirmek için damıtılmış BERT modeline nasıl ince ayar yapılacağını göreceksiniz. Dil modellerinin başka birçok büyüleyici uygulaması olmasına rağmen, basit ama klasik bir örnek olarak duygu sınıflandırmasını kasıtlı olarak seçtik. Ayrıca, tanıdık IMDb film incelemesi veri kümesini kullanarak, BERT modelini *Bölüm 8, Makine Öğrenimini Duygu Analizine Uygulama*'daki lojistik regresyon modeli ve *Bölüm 15*, *Tekrarlayan Sinir Ağları Kullanarak Sıralı Verileri Modelleme*'deki RNN ile karşılaştırarak BERT modelinin tahmin performansı hakkında iyi bir fikir edinebiliriz.

IMDb film inceleme veri kümesini yükleme

Bu alt bölümde, gerekli paketleri ve eğitme, doğrulama ve test kümelerine ayrılmış veri kümesini yükleyerek başlayacağız.

Bu eğitimin BERT ile ilgili bölümlerinde, *yeni metin oluşturmak için GPT-2'yi kullanma başlıklı* önceki bölümde kurduğumuz Hugging Face tarafından oluşturulan açık kaynaklı dönüştürücü kütüphanesini (https://huggingface.co/transformers/) kullanacağız.

Bu bölümde kullandığımız **DistilBERT** modeli, önceden eğitilmiş bir BERT temel modelinin damıtılmasıyla oluşturulan hafif bir transformatör modelidir. Orijinal eğitimsiz BERT temel modeli 110 milyondan fazla parametre içerirken DistilBERT yüzde 40 daha az parametreye sahiptir. Ayrıca, DistilBERT yüzde 60 daha hızlı çalışır ve BERT'in GLUE dil anlama kıyaslamasındaki performansının yüzde 95'ini hala korur.

Aşağıdaki kod, verileri hazırlamak ve DistilBERT modeline ince ayar yapmak için bu bölümde kullanacağımız tüm paketleri içe aktarır:

```
>> import gzip
>> import shutil
>>> import time

>> import pandas as pd
>>> talepleri içe aktar
>> import torch
>> import torch.nn.functional as F
>> import torchtext

>> import transformers
>> from transformers import DistilBertTokenizerFast
>> from transformers import DistilBertForSequenceClassification
```

Ardından, ağı eğittiğimiz epok sayısı, cihaz özellikleri ve rastgele tohum dahil olmak üzere bazı genel ayarları belirliyoruz. Sonuçları yeniden üretmek için 123 gibi belirli bir rastgele tohum ayarladığınızdan emin olun:

```
> >  torch.backends.cudnn.deterministic = True
>> RANDOM_SEED = 123
>> torch.manual_seed(RANDOM_SEED)
```

```
>>> DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

>> NUM_EPOCHS = 3
```

Daha önce *Bölüm 8* ve *Bölüm 9*'da gördüğünüz IMDb film inceleme veri kümesi üzerinde çalışacağız.

15. Aşağıdaki kod sıkıştırılmış veri kümesini getirir ve fermuarını açar:

```
>>> url = ("https://github.com/rasbt/"
...         "machine-learning-book/raw/"
...         "main/ch08/movie_data.csv.gz")
>>> dosya adı = url.split("/")[-1]

>> with open(dosya adı, "wb") as f:
...     r = requests.get(url)
...     f.write(r.content)

>> with gzip.open('movie_data.csv.gz', 'rb') as f_in:
...     with open('movie_data.csv', 'wb') as f_out:
...         shutil.copyfileobj(f_in, f_out)
```

Bölüm 8'deki movie_data.csv dosyası hala sabit diskinizdeyse, bu indirme ve açma prosedürünü atlayabilirsiniz.

Ardından, verileri bir pandas DataFrame'e yüklüyoruz ve doğru göründüğünden emin oluyoruz:

```
>> df = pd.read_csv('movie_data.csv')
>> df.head(3)
```

	review	sentiment
0	In 1974, the teenager Martha Moxley (Maggie Gr...	1
1	OK... so... I really like Kris Kristofferson a...	0
2	***SPOILER*** Do not read this, if you think a...	0

Şekil 16.17: IMDb film incelemesi veri kümesinin ilk üç satırı

Bir sonraki adım, veri setini ayrı eğitim, doğrulama ve test setlerine bölmektir. Burada, incelemelerin yüzde 70'ini eğitim seti için, yüzde 10'unu doğrulama seti için ve kalan yüzde 20'sini test için kullanıyoruz:

```
>>> train_texts = df.iloc[:35000]['review'].values
>> train_labels = df.iloc[:35000]['sentiment'].values

>> geçerli_metinler = df.iloc[35000:40000]['inceleme'].değerler
```

 alinpikbarbell@gmail.com

```
>>> valid_labels = df.iloc[35000:40000]['sentiment'].values

>>> test_texts = df.iloc[40000:]['review'].values
>>> test_labels = df.iloc[40000:]['sentiment'].values
```

Veri kümesini simgeleştirme

Şimdiye kadar eğitim, doğrulama ve test setleri için metinleri ve etiketleri elde ettik. Şimdi, önceden eğitilmiş model sınıfından miras alınan tokenizer uygulamasını kullanarak metinleri tek tek kelime tokenlarına ayıracağız:

```
>>> tokenizer = DistilBertTokenizerFast.from_pretrained(
...         'distilbert-base-uncased'
... )

>> train_encodings = tokenizer(list(train_texts), truncation=True, padding=True)
>> geçerli_kodlamalar = tokenizer(list(valid_texts), truncation=True, padding=True)
>> test_encodings = tokenizer(list(test_texts), truncation=True, padding=True)
```

Farklı tokenizer'lar seçme

Farklı türde belirteçler uygulamakla ilgileniyorsanız, Hugging Face tarafından oluşturulan ve bakımı yapılan belirteç paketini (https://huggingface.co/docs/tokenizers/python/latest/) keşfetmekten çekinmeyin. Bununla birlikte, kalıtsal belirteçleştiriciler önceden eğitilmiş model ile veri kümesi arasındaki tutarlılığı korur, bu da bizi modele karşılık gelen belirli belirteçleştiriciyi bulmak için fazladan çaba harcamaktan kurtarır. Başka bir deyişle, önceden eğitilmiş bir modele ince ayar yapmak istiyorsanız, devralınmış bir belirteçleyici kullanmak önerilen yaklaşımdır.

Son olarak, her şeyi IMDbDataset adlı bir sınıfta toplayalım ve ilgili veri yükleyicileri oluşturalım. Kendi kendini tanımlayan böyle bir veri kümesi sınıfı, DataFrame biçimindeki özel film inceleme veri kümemiz için ilgili tüm özellikleri ve işlevleri özelleştirmemize olanak tanır:

```
>> class IMDbDataset(torch.utils.data.Dataset):
...         def init (self, encodings, labels):
...                 self.encodings = encodings
...                 self.labels = labels

>>>         def getitem (self, idx):
...                 item = {anahtar: torch.tensor(val[idx])
...                         for key, val in self.encodings.items()}
...                 item['labels'] = torch.tensor(self.labels[idx])
```

```
...             iade öğesi

>>>         def len (self):
...             return len(self.labels)


>> train_dataset = IMDbDataset(train_encodings, train_labels)
>> valid_dataset = IMDbDataset(valid_encodings, valid_labels)
>> test_dataset = IMDbDataset(test_encodings, test_labels)

>> train_loader = torch.utils.data.DataLoader(
...         train_dataset, batch_size=16, shuffle=True)
>> valid_loader = torch.utils.data.DataLoader(
...         valid_dataset, batch_size=16, shuffle=False)
>> test_loader = torch.utils.data.DataLoader(
...         test_dataset, batch_size=16, shuffle=False)
```

Genel veri yükleyici kurulumu önceki bölümlerden tanıdık gelse de, dikkat çekici bir ayrıntı getitem yöntemindeki item değişkenidir. Daha önce ürettiğimiz kodlamalar, tokenize edilmiş metinler hakkında pek çok bilgi depolamaktadır. Sözlüğü item değişkenine atamak için kullandığımız sözlük anlama yoluyla, yalnızca en alakalı bilgileri çıkarıyoruz. Örneğin, ortaya çıkan sözlük girdileri input_ids (belirteçlere karşılık gelen kelime dağarcığından benzersiz tamsayılar), labels (sınıf etiketleri) ve attention_mask içerir. Burada attention_mask, modelin hangi belirteçlerle ilgilenmesi gerektiğini gösteren ikili değerlere (0'lar ve 1'ler) sahip bir tensördür. Özellikle, 0'lar diziyi eşit uzunlukta doldurmak için kullanılan ve model tarafından göz ardı edilen belirteçlere karşılık gelir; 1'ler ise gerçek metin belirteçlerine karşılık gelir.

Önceden eğitilmiş bir BERT modelini yükleme ve ince ayar yapma

Veri hazırlığını hallettikten sonra, bu alt bölümde, önceden eğitilmiş DistilBERT modelini nasıl yükleyeceğinizi ve yeni oluşturduğumuz veri kümesini kullanarak nasıl ince ayar yapacağınızı göreceksiniz. Önceden eğitilmiş modeli yüklemek için kod aşağıdaki gibidir:

```
> > >   model = DistilBertForSequenceClassification.from_pretrained(
...         'distilbert-base-uncased')
>> model.to(DEVICE)
>> model.train()

>>> optim = torch.optim.Adam(model.parameters(), lr=5e-5)
```

DistilBertForSequenceClassification, model üzerinde ince ayar yapmak istediğimiz aşağı akış görevini belirtir, bu durumda bu görev dizi sınıflandırmasıdır. Daha önce de belirtildiği gibi, 'distilbert-base-uncased', yönetilebilir boyut ve iyi performansa sahip BERT uncased temel modelinin hafif bir versiyonudur. "uncased" ifadesinin modelin büyük ve küçük harfler arasında ayrım yapmadığı anlamına geldiğini unutmayın.

 alinpikbarbell@gmail.com

Önceden eğitilmiş diğer transformatörleri kullanma

Transformers paketi ayrıca önceden eğitilmiş birçok başka model ve ince ayar için çeşitli aşağı akış görevleri sağlar. Bunlara https://huggingface.co/transformers/ adresinden göz atabilirsiniz.

Şimdi sıra modeli eğitmeye geldi. Bunu iki bölüme ayırabiliriz. İlk olarak, model performansını değerlendirmek için bir doğruluk fonksiyonu tanımlamamız gerekir. Bu doğruluk fonksiyonunun geleneksel sınıflandırma doğruluğunu hesapladığını unutmayın. Neden bu kadar ayrıntılı? Burada, büyük bir derin öğrenme modeliyle çalışırken RAM veya GPU bellek (VRAM) sınırlamalarını aşmak için veri kümesini toplu olarak yüklüyoruz:

```
>> def compute_accuracy(model, data_loader, device):
...         torch.no_grad() ile:
...             correct_pred, num_examples = 0, 0
...             for batch_idx, batch in enumerate(data_loader):
...                 ### Veri hazırla
...                 input_ids = batch['input_ids'].to(device)
...                 attention_mask = \
...                     batch['attention_mask'].to(device)
...                 etiketler = batch['etiketler'].to(cihaz)

...                 outputs = model(input_ids,
...                     attention_mask=attention_mask)
...                 logits = outputs['logits']
...                 predicted_labels = torch.argmax(logits, 1)
...                 num_examples += labels.size(0)
...                 correct_pred += \
...                         (predicted_labels == labels).sum()
...         return correct_pred.float()/num_examples * 100
```

compute_accuracy fonksiyonunda, belirli bir yığını yüklüyoruz ve ardından çıktılardan tahmin edilen etiketleri elde ediyoruz. Bunu yaparken num_examples aracılığıyla toplam örnek sayısını takip ediyoruz. Benzer şekilde, doğru tahminlerin sayısını da correct_pred değişkeni aracılığıyla takip ediyoruz. Son olarak, tüm veri kümesi üzerinde yineleme yaptıktan sonra, doğruluğu doğru tahmin edilen etiketlerin oranı olarak hesaplıyoruz.

Genel olarak, compute_accuracy fonksiyonu aracılığıyla, sınıf etiketlerini elde etmek için dönüştürücü modelini nasıl kullanabileceğimize dair bir fikir edinebilirsiniz. Yani, modeli input_ids ile birlikte burada bir belirtecin gerçek bir metin belirteci mi yoksa dizileri eşit uzunlukta doldurmak için kullanılan bir belirteç mi olduğunu gösteren attention_mask bilgisiyle besliyoruz. Model çağrısı daha sonra transformatör kütüphanesine özgü bir SequenceClassifierOutput nesnesi olan çıktıları döndürür. Bu nesneden, daha önceki bölümlerde yaptığımız gibi argmax fonksiyonu aracılığıyla sınıf etiketlerine dönüştürdüğümüz logitleri elde ederiz.

Son olarak, ana kısma gelelim: eğitim (ya da daha doğrusu ince ayar) döngüsü. Fark edeceğiniz gibi, *transformers* kütüphanesinden bir modele ince ayar yapmak, saf PyTorch'ta sıfırdan bir modeli eğitmeye çok benzer:

```
>> start_time = time.time()

>> for epoch in range(NUM_EPOCHS):

...     model.train()

...     for batch_idx, batch in enumerate(train_loader):

...         ### Veri hazırla
...         input_ids = batch['input_ids'].to(DEVICE)
...         attention_mask = batch['attention_mask'].to(DEVICE)
...         etiketler = batch['etiketler'].to(DEVICE)

...         ### İleri pas
...         outputs = model(input_ids,
...                         dikkat_maskesi=dikkat_maskesi,
...                         labels=labels)
...         loss, logits = outputs['loss'], outputs['logits']

...         ### Geriye doğru geçiş
...         optim.zero_grad()
...         loss.backward()
...         optim.step()

...         ### Günlük kaydı
...         if not batch_idx % 250:
...             print(f'Epoch: {epoch+1:04d}/{NUM_EPOCHS:04d}'
...                   f' | Batch'
...                   f'{batch_idx:04d}/'
...                   f'{len(train_loader):04d} | '
...                   f'Kayıp: {kayıp:.4f}')

...     model.eval()

...     torch.set_grad_enabled(False) ile:
...         print(f'Eğitim doğruluğu: '
...               f'{compute_accuracy(model, train_loader, DEVICE):.2f}%'
```

dilinpikbarbell@gmail.com

```
...                 f'\nGeçerli doğruluk: '
...                 f'{compute_accuracy(model, valid_loader, DEVICE):.2f}%')

...     print(f'Geçen süre: {(time.time() - start_time)/60:.2f} min')

... print(f'Toplam Eğitim Süresi: {(time.time() - start_time)/60:.2f} min')
... print(f'Test doğruluğu: {compute_accuracy(model, test_loader, DEVICE):.2f}%')
```

Yukarıdaki kod tarafından üretilen çıktı aşağıdaki gibidir (kodun tamamen deterministik olmadığını unutmayın, bu nedenle aldığınız sonuçlar biraz farklı olabilir):

```
Dönem: 0001/0003 | Toplu İş 0000/2188 | Kayıp: 0.6771
Dönem: 0001/0003 | Toplu İş 0250/2188 | Kayıp: 0.3006
Dönem: 0001/0003 | Toplu İş 0500/2188 | Kayıp: 0.3678
Dönem: 0001/0003 | Toplu İş 0750/2188 | Kayıp: 0.1487
Dönem: 0001/0003 | Toplu İş 1000/2188 | Kayıp: 0.6674
Dönem: 0001/0003 | Toplu İş 1250/2188 | Kayıp: 0.3264
Dönem: 0001/0003 | Toplu İş 1500/2188 | Kayıp: 0.4358
Dönem: 0001/0003 | Toplu İş 1750/2188 | Kayıp: 0.2579
Dönem: 0001/0003 | Toplu İş 2000/2188 | Kayıp: 0.2474
Eğitim doğruluğu: 96.32%


Dönem: 0002/0003 | Parti 0000/2188 | Kayıp: 0.0850
Dönem: 0002/0003 | Parti 0250/2188 | Kayıp: 0.3433
Dönem: 0002/0003 | Parti 0500/2188 | Kayıp: 0.0793
Dönem: 0002/0003 | Parti 0750/2188 | Kayıp: 0.0061
Dönem: 0002/0003 | Parti 1000/2188 | Kayıp: 0.1536
Dönem: 0002/0003 | Parti 1250/2188 | Kayıp: 0.0816
Dönem: 0002/0003 | Parti 1500/2188 | Kayıp: 0.0786
Dönem: 0002/0003 | Parti 1750/2188 | Kayıp: 0.1395
Dönem: 0002/0003 | Parti 2000/2188 | Kayıp: 0.0344
Eğitim doğruluğu: %98,35
Geçerli doğruluk: %92,46
Geçen süre: 41.41 dakika
Dönem: 0003/0003 | Parti 0000/2188 | Kayıp: 0.0403
Dönem: 0003/0003 | Parti 0250/2188 | Kayıp: 0.0036
Dönem: 0003/0003 | Parti 0500/2188 | Kayıp: 0.0156
Dönem: 0003/0003 | Parti 0750/2188 | Kayıp: 0.0114
```

6587191CA584E


```
Dönem: 0003/0003 | Parti    1000/2188  | Kayıp: 0.1227
Dönem: 0003/0003 | Parti    1250/2188  | Kayıp: 0.0125
```

alinpikbarbell@gmail.com

```
Dönem: 0003/0003 | Yığın 1500/2188 | Kayıp: 0,0074
Dönem: 0003/0003 | Yığın 1750/2188 | Kayıp:
0,0202 Dönem: 0003/0003 | Yığın 2000/2188 | Kayıp:
0,0746 Eğitim doğruluğu: %99,08
Geçerli doğruluk: %91,84
Geçen süre: 62,15 dakika
Toplam Eğitim Süresi: 62,15 dakika
Test doğruluğu: %92,50
```

Bu kodda, birden fazla epok üzerinde yineleme yapıyoruz. Her epokta aşağıdaki adımları gerçekleştiriyoruz:

1. Girdiyi üzerinde çalıştığımız cihaza yükleyin (GPU veya CPU)
2. Model çıktısını ve kaybı hesaplayın
3. Ağırlık parametrelerini kaybı geriye yayarak ayarlayın
4. Model performansını hem eğitim hem de doğrulama seti üzerinde değerlendirin

Eğitim süresinin farklı cihazlarda değişiklik gösterebileceğini unutmayın. Üç epoktan sonra, test veri kümesindeki doğruluk yaklaşık yüzde 93'e ulaşır; bu, RNN'nin *Bölüm 15*'te elde ettiği yüzde 85 test doğruluğuna kıyasla önemli bir gelişmedir.

Trainer API kullanarak bir transformatöre daha kolay ince ayar yapma

Önceki alt bölümde, bir transformatör modeline ince ayar yapmanın bir RNN veya CNN modelini sıfırdan eğitmekten çok da farklı olmadığını göstermek için PyTorch'taki eğitim döngüsünü manuel olarak uyguladık. Bununla birlikte, transformatör kütüphanesinin, bu alt bölümde tanıtacağımız Trainer API gibi ek kolaylık sağlayan birkaç güzel ekstra özellik içerdiğini unutmayın.

Hugging Face tarafından sağlanan Trainer API, çok çeşitli eğitim seçenekleri ve çeşitli yerleşik özelliklerle transformatör modelleri için optimize edilmiştir. Trainer API'yi kullanırken, kendi başımıza eğitim döngüleri yazma çabasını atlayabiliriz ve bir transformatör modelini eğitmek veya ince ayar yapmak bir fonksiyon (veya yöntem) çağrısı kadar basittir. Bunun pratikte nasıl çalıştığını görelim.

Önceden eğitilmiş modeli yükledikten sonra

```
>>> model = DistilBertForSequenceClassification.from_pretrained(
...     'distilbert-base-uncased')
>> model.to(DEVICE)
>> model.train();
```

Önceki bölümdeki eğitim döngüsü aşağıdaki kodla değiştirilebilir:

```
>>> optim = torch.optim.Adam(model.parameters(), lr=5e-5)

>> from transformers import Trainer, TrainingArguments
```

```
>> training_args = TrainingArguments(
...         output_dir='./results',
...         num_train_epochs=3,
...         per_device_train_batch_size=16,
...         per_device_eval_batch_size=16,
...         logging_dir='./logs',
...         logging_steps=10,
... )

>> eğitmen = Eğitmen(
...         model=model,
...         args=training_args,
...         train_dataset=train_dataset,
...         optimizers=(optim, None) # optim ve öğrenme oranı zamanlayıcısı
... )
```

Önceki kod parçacıklarında, ilk olarak girdi ve çıktı konumları, epok sayısı ve yığın boyutları ile ilgili nispeten kendi kendine planlayıcı ayarlar olan eğitim argümanlarını tanımladık. Ayarları mümkün olduğunca basit tutmaya çalıştık; ancak, birçok ek ayar mevcuttur ve ek ayrıntılar için TrainingArguments belge sayfasına başvurmanızı öneririz: https://huggingface.co/transformers/main_classes/trainer.html#trainingarguments.

Daha sonra bu TrainingArguments ayarlarını Trainer sınıfına aktararak yeni bir trainer nesnesi oluşturduk. Trainer'ı ayarlarla, ince ayar yapılacak modelle ve eğitim ve değerlendirme setleriyle başlattıktan sonra, trainer.train() yöntemini çağırarak modeli eğitebiliriz (bu yöntemi daha sonra kullanacağız). İşte bu kadar, Trainer API'sini kullanmak önceki kodda gösterildiği kadar basittir ve başka bir boilerplate koduna gerek yoktur.

Ancak, test veri kümesinin bu kod parçacıklarına dahil edilmediğini ve bu alt bölümde herhangi bir değerlendirme ölçütü belirtmediğimizi fark etmiş olabilirsiniz. Bunun nedeni, Trainer API'sinin yalnızca eğitim kaybını göstermesi ve varsayılan olarak eğitim süreci boyunca model değerlendirmesi sağlamamasıdır. Nihai model performansını görüntülemenin iki yolu vardır, bunları daha sonra göstereceğiz.

Nihai modeli değerlendirmek için ilk yöntem, başka bir Trainer örneği için compute_metrics argümanı olarak bir değerlendirme fonksiyonu tanımlamaktır. compute_metrics işlevi, modellerin test tahminleri üzerinde logitler (modelin varsayılan çıktısıdır) ve test etiketleri olarak çalışır. Bu fonksiyonu örneklemek için Hugging Face'in veri setleri kütüphanesini pip install datasets aracılığıyla yüklemenizi ve aşağıdaki şekilde kullanmanızı öneririz:

```
>> from datasets import load_metric
>> import numpy as np

>>> metric = load_metric("accuracy")

>> def compute_metrics(eval_pred):
```

 dilinpikbarbell@gmail.com

```
...         logits, labels = eval_pred
...         # not: logits bir numpy dizisidir, pytorch tensörü değildir
...         predictions = np.argmax(logits, axis=-1)
...         return metric.compute(
...             tahminler=predictions, referanslar=labels)
```

Güncellenmiş Trainer örneklemesi (şimdi compute_metrics dahil) aşağıdaki gibidir:

```
>> trainer=Trainer(
...         model=model,
...         args=training_args,
...         train_dataset=train_dataset,
...         eval_dataset=test_dataset,
...         compute_metrics=compute_metrics,
...         optimizers=(optim, None) # optim ve öğrenme oranı zamanlayıcısı
... )
```

Şimdi modeli eğitelim (kodun tamamen deterministik olmadığını ve bu yüzden biraz farklı sonuçlar alabileceğinizi unutmayın):

```
>> start_time = time.time()
>> trainer.train()

***** Koşu eğitimi ***** Örnek
   sayısı = 35000
   Num Epochs = 3
   Cihaz başına anlık parti boyutu = 16
   Toplam tren yığın boyutu (paralel, dağıtılmış ve biriktirme) = 16 Gradyan
   Biriktirme adımları = 1
   Toplam optimizasyon adımı = 6564

Adım   Eğitim Kaybı
10     0.705800
20     0.684100
30     0.681500
40     0.591600
50     0.328600
60     0.478300
...


>> print(f'Toplam Eğitim Süresi: '
...         f'{(time.time() - start_time)/60:.2f} min')
Toplam Eğitim Süresi: 45,36 dk
```

GPU'nuza bağlı olarak bir saate kadar sürebilen eğitim tamamlandıktan sonra test seti üzerinde model performansını elde etmek için trainer.evaluate():

```
>> print(trainer.evaluate())

***** Çalışan Değerlendirme *****
Örnek sayısı = 10000
Parti boyutu = 16
100%|""""""""""""""""""""""""""""""""""""""""""""""""""""""""| 625/625 [10:59<00:00, 1.06s/
o]
{'eval_loss': 0.30534815788269043,
 'eval_accuracy': 0.9327,
 'eval_runtime': 87.1161,
 'eval_samples_per_second': 114.789,
 'eval_steps_per_second': 7.174,
 'epoch': 3.0}
```

Gördüğümüz gibi, değerlendirme doğruluğu daha önce kullandığımız Py- Torch eğitim döngüsüne benzer şekilde yüzde 94 civarındadır. (Eğitim adımını atladığımızı unutmayın, çünkü model önceki trainer.train() çağrısından sonra zaten ince ayarlanmıştır). Manuel eğitim yaklaşımımız ile Trainer sınıfını kullanmak arasında küçük bir tutarsızlık vardır, çünkü Trainer sınıfı bazı farklı ve bazı ek ayarlar kullanır.

Nihai test seti doğruluğunu hesaplamak için kullanabileceğimiz ikinci yöntem, önceki bölümde tanımladığımız compute_ accuracy fonksiyonumuzu yeniden kullanmaktır. Aşağıdaki kodu çalıştırarak ince ayarlı modelin test veri kümesi üzerindeki performansını doğrudan

```
>> model.eval()
>> model.to(DEVICE)

>> print(f'Test doğruluğu: {compute_accuracy(model, test_loader,
DEVICE):.2f}%')

Test doğruluğu: %93,27
```

değerlendirebiliriz:

Aslında, eğitim sırasında modelin performansını düzenli olarak kontrol etmek istiyorsanız, eğitim argümanlarını aşağıdaki gibi tanımlayarak eğiticinin her epoktan sonra model değerlendirmesini yazdırmasını isteyebilirsiniz:

```
>> from transformers import TrainingArguments

>> training_args = TrainingArguments("test_trainer",
...        evaluation_strategy="epoch", ...)
```

 alinpikbarbell@gmail.com

Ancak, hiperparametreleri değiştirmeyi veya optimize etmeyi ve ince ayar prosedürünü birkaç kez tekrarlamayı planlıyorsanız, test setini bağımsız tutmak için bu amaçla doğrulama setini kullanmanızı öneririz. Bunu, Trainer'ı valid_dataset kullanarak örneklendirerek başarabiliriz:

```
>> trainer=Trainer(
...         model=model,
...         args=training_args,
...         train_dataset=train_dataset,
...         eval_dataset=valid_dataset,
...         compute_metrics=compute_metrics,
... )
```

Bu bölümde, sınıflandırma için bir BERT modeline nasıl ince ayar yapabileceğimizi gördük. Bu, genellikle sıfırdan eğittiğimiz RNN'ler gibi diğer derin öğrenme mimarilerini kullanmaktan farklıdır. Ancak, araştırma yapmadığımız ve yeni transformatör mimarileri geliştirmeye çalışmadığımız sürece (çok pahalı bir çaba) transformatör modellerini önceden eğitmek gerekli değildir. Transformatör modelleri genel, etiketsiz veri kümesi kaynakları üzerinde eğitildiğinden, bunları önceden eğitmek zamanımızı ve kaynaklarımızı iyi kullanmak olmayabilir; ince ayar yapmak gitmenin yoludur.

Özet

Bu bölümde, doğal dil işleme için yepyeni bir model mimarisi olan dönüştürücü mimarisini tanıttık. Dönüştürücü mimarisi öz dikkat adı verilen bir kavram üzerine inşa edilmiştir ve bu kavramı adım adım tanıtmaya başladık. İlk olarak, uzun cümleler için çeviri yeteneklerini geliştirmek amacıyla dikkat ile donatılmış bir RNN'ye baktık. Ardından, öz dikkat kavramını nazikçe tanıttık ve dönüştürücü içindeki çok kafalı dikkat modülünde nasıl kullanıldığını açıkladık.

Orijinal transformatörün 2017'de yayınlanmasından bu yana transformatör mimarisinin birçok farklı türevi ortaya çıkmış ve gelişmiştir. Bu bölümde, en popüler olanlardan bazılarına odaklandık: GPT model ailesi, BERT ve BART. GPT, özellikle yeni metin oluşturmada iyi olan tek yönlü bir modeldir. BERT, sınıflandırma gibi diğer görev türleri için daha uygun olan çift yönlü bir yaklaşım benimser. Son olarak, BART hem BERT'in çift yönlü kodlayıcısını hem de GPT'nin tek yönlü kod çözücüsünü birleştirir. İlgilenen okuyucular, aşağıdaki iki araştırma makalesi aracılığıyla ek transformatör tabanlı mimariler hakkında bilgi edinebilirler:

1. *Doğal Dil İşleme için Önceden Eğitilmiş Modeller: Qiu* ve meslektaşları tarafından yapılan *bir araştırma,* 2020. https://arxiv.org/abs/2003.08271 adresinden temin edilebilir.
2. *AMMUS : Doğal Dil İşlemede Transformatör Tabanlı Önceden Eğitilmiş Modeller* Üzerine *Bir Araştırma Kayan* ve meslektaşları, 2021. https://arxiv.org/abs/2108.05542 adresinde mevcuttur.

Dönüştürücü modeller genellikle RNN'lerden daha fazla veriye ihtiyaç duyar ve ön eğitim için büyük miktarlarda veriye ihtiyaç duyar. Ön eğitim, daha sonra daha küçük etiketli veri kümeleri üzerinde ince ayar yapılarak belirli görevler için özelleştirilebilecek genel bir dil modeli oluşturmak için büyük miktarlarda etiketsiz veriden yararlanır.

6587191CA584E

Bunun pratikte nasıl çalıştığını görmek için, Hugging Face'ten önceden eğitilmiş bir BERT modeli indirdik
transformers kütüphanesi ve IMDb film incelemesi veri kümesinde duyarlılık sınıflandırması için ince ayar yaptı.

Bir sonraki bölümde, üretken karşıt ağları tartışacağız. Adından da anlaşılacağı gibi, üretken düşman ağları, bu bölümde tartıştığımız GPT modellerine benzer şekilde, yeni veri üretmek için kullanılabilen modellerdir. Ancak, şimdi doğal dil modelleme konusunu arkamızda bırakıyoruz ve üretken düşman ağlarını bilgisayarla görme ve bu ağların orijinal olarak tasarlandığı görev olan yeni görüntüler üretme bağlamında inceleyeceğiz.

Kitabımızın Discord alanına katılın

Benzer düşünen insanlarla tanışmak ve 2000'den fazla üyemizle birlikte öğrenmek için Discord topluluğumuza katılın:
https://packt.link/MLwPyTorch

6587191CA584E
olimpikbarbell@gmail.com

olimpikbarbell@gmail.com

17

Yeni Verilerin Sentezlenmesi için Üretken Çekişmeli Ağlar

Bir önceki bölümde, dizileri modellemek için **tekrarlayan sinir ağlarına** odaklandık. Bu bölümde, **üretken karşıt ağları** (**GAN'lar**) keşfedeceğiz ve yeni veri örneklerinin sentezlenmesindeki uygulamalarını göreceğiz. GAN'lar, bilgisayarların yeni veriler (yeni görüntüler gibi) üretmesine olanak tanıyan derin öğrenme alanındaki en önemli buluşlardan biri olarak kabul edilmektedir.

Bu bölümde aşağıdaki konuları ele alacağız:

- Yeni verilerin sentezlenmesi için üretken modellerin tanıtılması
- Otomatik kodlayıcılar, varyasyonel otomatik kodlayıcılar ve bunların GAN'larla ilişkisi
- GAN'ların yapı taşlarını anlama
- El yazısı rakamları oluşturmak için basit bir GAN modeli uygulama
- Dönüştürülmüş evrişim ve toplu normalleştirmeyi anlama
- GAN'ların iyileştirilmesi: derin evrişimli GAN'lar ve Wasserstein mesafesini kullanan GAN'lar

Üretken çekişmeli ağların tanıtılması

Öncelikle GAN modellerinin temellerine bakalım. Bir GAN'ın genel amacı, eğitim veri kümesiyle aynı dağılıma sahip yeni veriler sentezlemektir. Bu nedenle, GAN'lar, orijinal halleriyle, etiketli veri gerekmediğinden, makine öğrenimi görevlerinin denetimsiz öğrenme kategorisinde kabul edilir. Bununla birlikte, orijinal GAN'a yapılan uzantıların hem yarı denetimli hem de denetimli alanlarda yer alabileceğini belirtmek gerekir.

Genel GAN kavramı ilk olarak 2014 yılında Ian Goodfellow ve meslektaşları tarafından derin **sinir ağları** (**NN'ler**) kullanarak yeni görüntüler sentezlemek için bir yöntem olarak önerilmiştir (*Generative Adversarial Nets*, *Advances in Neural Information Processing Systems* içinde, *I. Goodfellow*, *J. Pouget-Abadie, M. Mirza*, *B. Xu, D. Warde-Farley, S. Ozair, A. Courville* ve *Y. Bengio,* s. 2672-2680, 2014). Bu makalede önerilen ilk GAN mimarisi, çok katmanlı algılayıcı mimarilerine benzer şekilde tam bağlı katmanlara dayansa ve düşük çözünürlüklü MNIST benzeri el yazısı rakamları üretmek için eğitilse de, bu yeni yaklaşımın uygulanabilirliğini göstermek için daha çok bir kavram kanıtı olarak hizmet etmiştir.

Bununla birlikte, tanıtılmasından bu yana, orijinal yazarların yanı sıra diğer birçok araştırmacı, mühendislik ve bilimin farklı alanlarında çok sayıda iyileştirme ve çeşitli uygulamalar önermiştir; örneğin, bilgisayarla görmede, GAN'lar görüntüden görüntüye çeviri (bir giriş görüntüsünün bir çıkış görüntüsüne nasıl eşleneceğini öğrenme), görüntü süper çözünürlüğü (düşük çözünürlüklü bir sürümden yüksek çözünürlüklü bir görüntü oluşturma), görüntü boyama (bir görüntünün eksik kısımlarının nasıl yeniden yapılandırılacağını öğrenme) ve daha birçok uygulama için kullanılır. Örneğin, GAN araştırmalarındaki son gelişmeler, yeni, yüksek çözünürlüklü yüz görüntüleri oluşturabilen modellere yol açmıştır. Bu tür yüksek çözünürlüklü görüntülerin örnekleri, bir GAN tarafından üretilen sentetik yüz görüntülerinin sergilendiği https://www.thispersondoesnotexist.com/ adresinde bulunabilir.

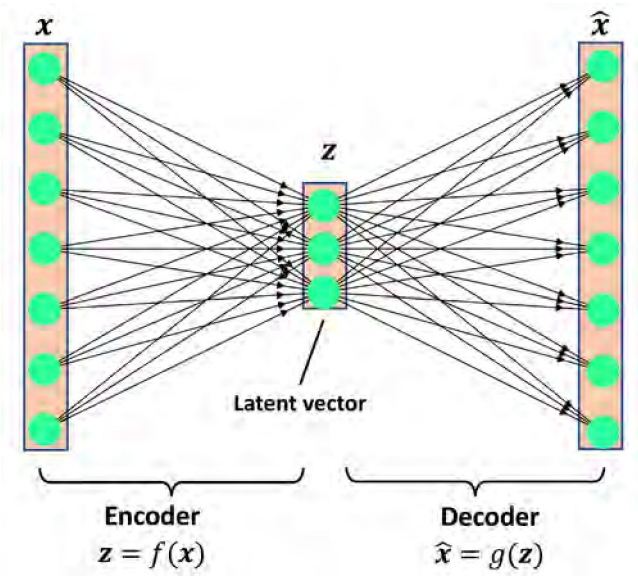
Otomatik kodlayıcılarla başlamak

GAN'ların nasıl çalıştığını tartışmadan önce, ilk olarak eğitim verilerini sıkıştırabilen ve açabilen oto kodlayıcılarla başlayacağız. Standart oto kodlayıcılar yeni veri oluşturamazken, işlevlerini anlamak bir sonraki bölümde GAN'larda gezinmenize yardımcı olacaktır.

Otomatik kodlayıcılar bir araya getirilmiş iki ağdan oluşur: bir kodlayıcı ağı ve bir kod **çözücü** ağı. Kodlayıcı ağ, $\boldsymbol{x}$ örneği ile ilişkili d *boyutlu* bir giriş özellik vektörü alır (yani, $\boldsymbol{xx}\ \boldsymbol{x}\ R^d$) ve bunu p *boyutlu* bir vektör olan $\boldsymbol{z'}$***ye kodlar*** (yani, $z\ \boldsymbol{z}\ R^p$). Diğer bir deyişle

Diğer bir deyişle, kodlayıcının rolü $\boldsymbol{z} = f(\boldsymbol{x})$ fonksiyonunun nasıl modelleneceğini öğrenmektir. Kodlanmış vektör, z,

gizli vektör veya gizli özellik temsili olarak da adlandırılır. Tipik olarak, gizli vektörün boyutluluğu giriş örneklerinin boyutluluğundan daha azdır; başka bir deyişle, $p < d$. Bu nedenle, kodlayıcının bir veri sıkıştırma işlevi gördüğünü söyleyebiliriz. Daha sonra, kod çözücü $\boldsymbol{x\hat{x}}$'yi düşük boyutlu gizli vektör $\boldsymbol{z'}$den açar, burada kod çözücüyü bir fonksiyon olarak düşünebiliriz, $\boldsymbol{x\hat{x}} = g(z)$. Basit bir otomatik kodlayıcı mimarisi *Şekil 17.1'de* gösterilmektedir; burada kodlayıcı ve kod çözücü kısımlarının her biri yalnızca bir tam bağlı katmandan oluşmaktadır:



Şekil 17.1: Bir otomatik kodlayıcının mimarisi

 alinpikbarbell@gmail.com

Otomatik kodlayıcılar ve boyut indirgeme arasındaki bağlantı

Bölüm 5, *Boyutluluk Azaltma Yoluyla Veri Sıkıştırma'*da, **temel bileşen analizi** (**PCA**) ve **doğrusal diskriminant analizi (LDA)** gibi boyutluluk azaltma tekniklerini öğrendiniz. Otomatik kodlayıcılar da bir boyutsallık azaltma tekniği olarak kullanılabilir. Aslında, iki alt ağdan (kodlayıcı ve kod çözücü) herhangi birinde doğrusal olmayan bir durum olmadığında, otomatik kodlayıcı yaklaşımı PCA ile *neredeyse aynıdır.*

Bu durumda, tek katmanlı bir kodlayıcının ağırlıklarının (gizli katman ve doğrusal olmayan aktivasyon fonksiyonu yok) U matrisi ile gösterildiğini varsayarsak, kodlayıcı $\boldsymbol{z} = \boldsymbol{U}^T$ x modelini oluşturur. Benzer şekilde, tek katmanlı bir doğrusal kod çözücü $\hat{\boldsymbol{x}}\ \boldsymbol{x} = Uz$ modelini oluşturur. Bu iki bileşeni bir araya getirdiğimizde, $\hat{\boldsymbol{x}}\ \boldsymbol{x} = UU^T\boldsymbol{xx}$ elde ederiz. Bu, PCA'nın ek bir ortonormal kısıtlamaya sahip olması dışında tam olarak PCA'nın yaptığı şeydir: $\boldsymbol{UU}^T = I_{n\times n}$.

Şekil 17.1 kodlayıcı ve kod çözücü içinde gizli katmanları olmayan bir oto kodlayıcıyı gösterirken, elbette daha etkili veri sıkıştırma ve yeniden yapılandırma işlevlerini öğrenebilen derin bir oto kodlayıcı oluşturmak için doğrusal olmayan (çok katmanlı bir NN'de olduğu gibi) birden fazla gizli katman ekleyebiliriz. Ayrıca, bu bölümde bahsedilen otomatik kodlayıcının tam bağlantılı katmanlar kullandığını unutmayın. Ancak görüntülerle çalışırken, *Bölüm 14, Derin Evrişimsel Sinir Ağlarıyla Görüntüleri Sınıflandırma'*da öğrendiğiniz gibi, tam bağlı katmanları evrişimsel katmanlarla değiştirebiliriz.

Gizli uzayın boyutuna dayalı diğer oto kodlayıcı türleri

Daha önce de belirtildiği gibi, bir otoenkoderin gizli uzayının boyutluluğu tipik olarak girdilerin boyutluluğundan daha düşüktür ($p < d$), bu da otoenkoderleri boyut azaltma için uygun hale getirir. Bu nedenle, gizli vektör genellikle "darboğaz" olarak da adlandırılır ve bir oto kodlayıcının bu özel yapılandırması da **eksik tamamlanmış olarak** adlandırılır. Bununla birlikte, gizli vektör z'nin boyutluluğunun aslında girdi örneklerinin boyutluluğundan daha büyük olduğu ($p > d$) **aşırı tamamlanmış olarak** adlandırılan farklı bir oto kodlayıcı kategorisi vardır.

Eksiksiz bir otomatik kodlayıcıyı eğitirken, kodlayıcı ve kod çözücünün basitçe giriş özelliklerini çıkış katmanlarına kopyalamayı (ezberlemeyi) öğrenebileceği önemsiz bir çözüm vardır. Açıkçası, bu çözüm çok kullanışlı değildir. Bununla birlikte, eğitim prosedüründe yapılacak bazı değişikliklerle, aşırı tamamlanmış otomatik kodlayıcılar *gürültü azaltma* için kullanılabilir.

Bu durumda, eğitim sırasında giriş örneklerine rastgele gürültü, $\epsilon\epsilon$, eklenir ve ağ gürültülü sinyalden, $\boldsymbol{xx}+\boldsymbol{xx}$, temiz örneği, x, yeniden oluşturmayı öğrenir. Ardından, değerlendirme zamanında, mevcut gürültüyü bu örneklerden çıkarmak için doğal olarak gürültülü olan (yani, ek yapay gürültü, $\epsilon\epsilon$, eklenmeyecek şekilde gürültü zaten mevcut olan) yeni örnekler sunuyoruz. Bu özel otomatik kodlayıcı mimarisi ve eğitim yöntemi, *denoising otomatik kodlayıcı* olarak adlandırılır.

Eğer ilgileniyorsanız, *Stacked denoising autoencoders* adlı araştırma makalesinde bu konuda daha fazla bilgi edinebilirsiniz: *Pascal Vincent* ve meslektaşları tarafından *yerel*

6587191CA584E

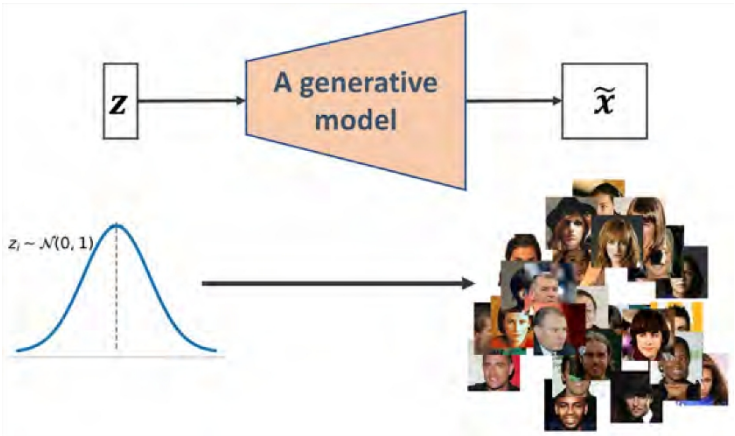
bir denoising kriteri ile derin bir ağda faydalı temsillerin öğrenilmesi, 2010 (http://www.jmlr.org/papers/v11/vincent10a. html).

 olinpikbarbell@gmail.com

Yeni verilerin sentezlenmesi için üretken modeller

Otomatik kodlayıcılar deterministik modellerdir, yani bir otomatik kodlayıcı eğitildikten sonra, bir girdi, $\boldsymbol{x}$, verildiğinde, girdiyi sıkıştırılmış versiyonundan daha düşük boyutlu bir uzayda yeniden oluşturabilecektir. Bu nedenle, sıkıştırılmış temsilin dönüştürülmesi yoluyla girdisini yeniden yapılandırmanın ötesinde yeni veri üretemez.

Öte yandan, üretken bir model, rastgele bir vektör olan $\boldsymbol{z}$***'den*** (gizli temsile karşılık gelen) yeni bir örnek, $\boldsymbol{x}\tilde{\boldsymbol{x}}$, üretebilir. Üretken bir modelin şematik gösterimi aşağıdaki şekilde gösterilmektedir. Rastgele vektör, z, özellikleri tamamen bilinen bir dağılımdan gelir, bu nedenle böyle bir dağılımdan kolayca örnek alabiliriz. Örneğin, $\boldsymbol{z}'$nin her bir elemanı [-1, 1] aralığındaki tekdüze dağılımdan (bunun için $z_i \sim Tekdüze(-1, 1)$ yazıyoruz) veya standart bir normal dağılımdan (bu durumda $z_i \sim Normal(\mu\ \mu\ \mu\ \sigma^2\ \mu\ 1)$ yazıyoruz) gelebilir:



Şekil 17.2: Üretken bir model

Dikkatimizi otomatik kodlayıcılardan üretken modellere kaydırdığımız için, bir otomatik kodlayıcının kod çözücü bileşeninin bir üretken model ile bazı benzerlikleri olduğunu fark etmiş olabilirsiniz. Özellikle, her ikisi de girdi olarak bir gizli vektör, z, alır ve $\boldsymbol{x}$ ile aynı uzayda bir çıktı döndürür. (Otomatik kodlayıcı için, $\boldsymbol{x}\hat{\boldsymbol{x}}$ bir girdinin, $\boldsymbol{x}$, yeniden yapılandırılmasıdır ve üretken model için, $\boldsymbol{x}\tilde{\boldsymbol{x}}$ sentezlenmiş bir örnektir).

Ancak, ikisi arasındaki en büyük fark, otomatik kodlayıcıda $\boldsymbol{z}'$nin dağılımını bilmememiz, üretken bir modelde ise z'nin dağılımının tamamen karakterize edilebilir olmasıdır. Yine de bir otomatik kodlayıcıyı üretken bir modele genelleştirmek mümkündür. Bir yaklaşım **varyasyonel otomatik kodlayıcıdır** (**VAE**).

Bir girdi örneği, $\boldsymbol{x}$, alan bir VAE'de, kodlayıcı ağ, gizli vektörün dağılımının iki momentini birleştirecek şekilde değiştirilir: ortalama, $\boldsymbol{\mu}$ ve varyans, $\boldsymbol{\sigma}^2$. Bir VAE'nin eğitimi sırasında, ağ bu momentleri standart bir normal dağılımınkilerle (yani sıfır ortalama ve birim varyans) eşleştirmeye zorlanır. VAE modeli eğitildikten sonra kodlayıcı atılır ve "öğrenilmiş" Gauss dağılımından rastgele $\boldsymbol{z}$ vektörleri besleyerek yeni örnekler, $\boldsymbol{x}\tilde{\boldsymbol{x}}$, üretmek için kod çözücü ağı kullanabiliriz.

VAE'lerin yanı sıra, *otoregresif modeller* ve *nor- malize akış modelleri* gibi başka türde üretken modeller de vardır. Ancak, bu bölümde sadece derin öğrenmedeki en yeni ve en popüler üretken model türlerinden biri olan GAN modellerine odaklanacağız.

Üretken model nedir?

Üretken modellerin geleneksel olarak veri girdi dağılımlarını, $p(x)$ veya girdi verilerinin ve ilişkili hedeflerin ortak dağılımlarını, $p(x, y)$ modelleyen algoritmalar olarak tanımlandığını unutmayın. Tanım gereği, bu modeller aynı zamanda **koşullu çıkarım** olarak bilinen başka bir özelliğe ($_{xj}$) bağlı olarak bazı özelliklerden ($_{xi}$) örnekleme yapabilmektedir. Ancak derin öğrenme bağlamında, **üretici model** terimi tipik olarak gerçekçi görünümlü veriler üreten modellere atıfta bulunmak için kullanılır. Bu, $p(x)$ girdi dağılımlarından örnekleme yapabileceğimiz, ancak koşullu çıkarım yapamayacağımız anlamına gelir.

GAN'lar ile yeni örnekler oluşturma

GAN'ların kısaca ne yaptığını anlamak için, öncelikle bilinen bir dağılımdan örneklenen rastgele bir vektör, ***z,*** alan ve bir çıktı görüntüsü, ***x, üreten bir*** ağımız olduğunu varsayalım. Bu ağa **jeneratör** (G) diyeceğiz ve üretilen çıktıya atıfta bulunmak için $\boldsymbol{x}\tilde{\boldsymbol{x}} = G(z)$ notasyonunu kullanacağız. Amacımızın bazı görüntüler, örneğin yüz görüntüleri, bina görüntüleri, hayvan görüntüleri ve hatta MNIST gibi el yazısı rakamlar üretmek olduğunu varsayalım.

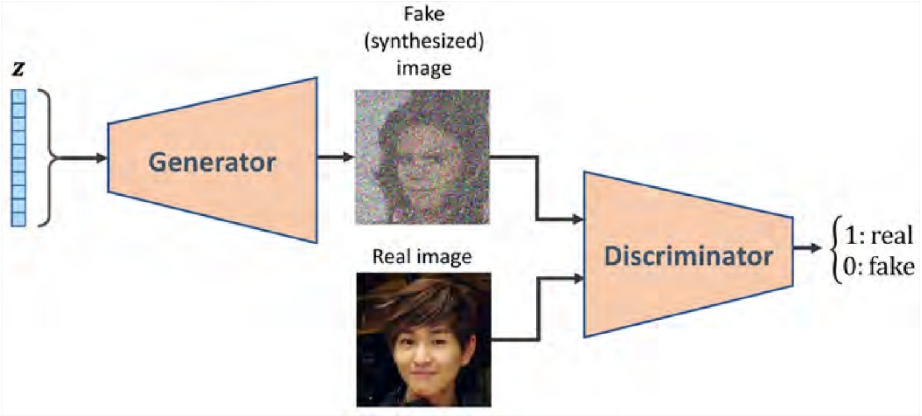
Her zaman olduğu gibi, bu ağı rastgele ağırlıklarla başlatacağız. Bu nedenle, bu ağırlıklar ayarlanmadan önceki ilk çıktı görüntüleri beyaz gürültü gibi görünecektir. Şimdi, görüntülerin kalitesini değerlendirebilen bir fonksiyon olduğunu düşünün (buna *değerlendirici fonksiyon* diyelim).

Eğer böyle bir fonksiyon mevcutsa, bu fonksiyondan gelen geri bildirimi jeneratör ağımıza üretilen görüntülerin kalitesini artırmak için ağırlıklarını nasıl ayarlayacağını söylemek için kullanabiliriz. Bu şekilde, jeneratörü bu değerlendirme fonksiyonundan gelen geri bildirime göre eğitebiliriz, böylece jeneratör çıktısını gerçekçi görünen görüntüler üretmeye doğru geliştirmeyi öğrenir.

Önceki paragrafta açıklandığı gibi bir değerlendirme fonksiyonu görüntü oluşturma görevini çok kolaylaştıracak olsa da, asıl soru görüntülerin kalitesini değerlendirmek için böyle evrensel bir fonksiyonun var olup olmadığı ve varsa nasıl tanımlandığıdır. Açıkçası, insanlar olarak, ağın çıktılarını gözlemlediğimizde çıktı görüntülerinin kalitesini kolayca değerlendirebiliriz; ancak sonucu beynimizden ağa geri yayamayız (henüz). Şimdi, eğer beynimiz sentezlenen görüntülerin kalitesini değerlendirebiliyorsa, aynı şeyi yapmak için bir NN modeli tasarlayabilir miyiz? Aslında GAN'ın genel fikri de budur.

 olimpikbarbell@gmail.com

Şekil 17.3'te gösterildiği gibi, bir GAN modeli, sentezlenmiş bir görüntüyü, $\boldsymbol{x}\tilde{\boldsymbol{x}}$, gerçek bir görüntüden, $\boldsymbol{x}$, tespit etmeyi öğrenen bir sınıflandırıcı olan **diskriminatör** (D) adı verilen ek bir NN'den oluşur:



Şekil 17.3: Ayırıcı, gerçek görüntü ile üreteç tarafından oluşturulan görüntüyü birbirinden ayırır

Bir GAN modelinde, iki ağ, jeneratör ve diskriminatör, birlikte eğitilir. İlk başta, model ağırlıklarını başlattıktan sonra, jeneratör gerçekçi görünmeyen görüntüler oluşturur. Benzer şekilde, ayırt edici de gerçek görüntüler ile üretici tarafından sentezlenen görüntüleri ayırt etme konusunda zayıf bir iş çıkarır. Ancak zamanla (yani eğitim yoluyla), her iki ağ da birbirleriyle etkileşime girdikçe daha iyi hale gelir. Aslında, iki ağ, jeneratörün ayırt ediciyi kandırabilmek için çıktısını geliştirmeyi öğrendiği düşmanca bir oyun oynar. Aynı zamanda, ayırıcı sentezlenmiş görüntüleri tespit etmede daha iyi hale gelir.

Bir GAN modelinde üreteç ve ayırıcı ağların kayıp fonksiyonlarının anlaşılması

GAN'ların amaç fonksiyonu, *I. Goodfellow* ve arkadaşlarının *Generative Adversarial Nets* adlı orijinal makalesinde (https://papers.nips.cc/paper/5423-generative-adversarial-nets. pdf) açıklandığı gibi aşağıdaki gibidir:

$$VVV(D), VV(G)) = E_{xx\,data}(xx)[\log D(xx)] + E_{zx\,pz(z)}[\log(1 - DVG(z))]$$

Burada, $VVV^{(D)}$, $VV^{(G)}$) **değer fonksiyonu** olarak adlandırılır ve bir getiri olarak yorumlanabilir: **değerini** ayırt ediciye (D) göre en üst düzeye çıkarmak ve değerini üreteci (G), yani $\min_G \max_D VVV^{(D)}, VV^{(G)})$. $D(\boldsymbol{x})$, girdinin girdiden farklı olup olmadığını gösteren olasılıktır. örnek, $\boldsymbol{x}$, gerçek veya sahte (yani üretilmiş). $E\boldsymbol{xxx}_{p\,data}(xx)[\log D(\boldsymbol{xx})]$ ifadesi beklenen veri dağılımındaki örneklere göre parantez içindeki miktarın değeri (dağılım gerçek örneklerin); $E_{zzp\,z(z)}[\log(1 - DDG(z))]$ girdi, $\boldsymbol{z}$, vektörlerinin dağılımına göre miktarın beklenen değerini ifade eder.

Böyle bir değer fonksiyonuna sahip bir GAN modelinin bir eğitim adımı iki optimizasyon adımı gerektirir: (1) ayırıcı için getiriyi maksimize etmek ve (2) üreteç için getiriyi minimize etmek. GAN'ları eğitmenin pratik bir yolu, bu iki optimizasyon adımı arasında geçiş yapmaktır: (1) bir ağın parametrelerini sabitlemek (dondurmak) ve diğerinin ağırlıklarını optimize etmek ve (2) ikinci ağı sabitlemek ve birincisini optimize etmek. Bu işlem her eğitim iterasyonunda tekrarlanmalıdır. Üreteç ağının sabit olduğunu ve ayırıcıyı optimize etmek istediğimizi varsayalım. Değerdeki her iki terim

fonksiyonu $VVVV^{(D)}$, $VV^{(G)}$) diskriminatörün optimize edilmesine katkıda bulunur; burada ilk terim gerçek örneklerle ilişkili kayıp ve ikinci terim sahte örnekler için kayıptır. Bu nedenle, G sabit olduğunda, amacımız $VVVV^{(D)}$, $VV^{(G)}$ *) maksimize* etmektir, bu da kriminatörün gerçek ve üretilmiş görüntüleri ayırt etmede daha iyi olmasını sağlamak anlamına gelir.

Gerçek ve sahte örnekler için kayıp terimlerini kullanarak ayırıcıyı optimize ettikten sonra, ayırıcıyı sabitler ve üreteci optimize ederiz. Bu durumda, $VVVV^{(D)}$, $VV^{(G)}$ 'deki yalnızca ikinci terim üretecin gradyanlarına katkıda bulunur. Sonuç olarak, D sabit olduğunda, amacımız $VVVV^{(D)}$, $VV^{(G)}$)'yi *minimize* etmektir,

$\min_{G} E_{z \sim p_z(z)} [\log (1 - D(G(z)))]$ şeklinde yazılabilir.] Orijinal GAN makalesinde belirtildiği gibi Goodfellow ve meslektaşları tarafından, bu fonksiyon, $\log (1 - D(G(z)))$, erken eğitim aşamalarında kaybolan gradyanlardan muzdariptir. Bunun nedeni, öğrenme sürecinin başlarında $G(z)$ çıktılarının gerçek örneklere benzememesi ve bu nedenle $D(G(z))$'nin yüksek güvenle sıfıra yakın olmasıdır. Bu olguya **doygunluk** denir. Bu sorunu çözmek için minimizasyon hedefini yeniden formüle edebiliriz,

$\min_{G} E_{z \sim p_z(z)} [\log (1 - D(G(z)))]$, $\max_{G} E_{z \sim p_z(z)} [\log (D(G(z)))]$ şeklinde yeniden yazarak.]

Bu değiştirme, üreteci eğitmek için gerçek ve sahte örneklerin etiketlerini değiştirebileceğimiz ve düzenli bir fonksiyon minimizasyonu gerçekleştirebileceğimiz anlamına gelir. Başka bir deyişle, üreteç tarafından sentezlenen örnekler sahte olsa ve bu nedenle 0 etiketli olsa da, bu örneklere 1 etiketi atayarak etiketleri tersine çevirebilir ve ikili çapraz entropi kaybını bu yeni etiketlerle en *aza* indirebiliriz.

$\max_{G} E_{z \sim p_z(z)} [\log (D(G(z)))]$ maksimize eder.

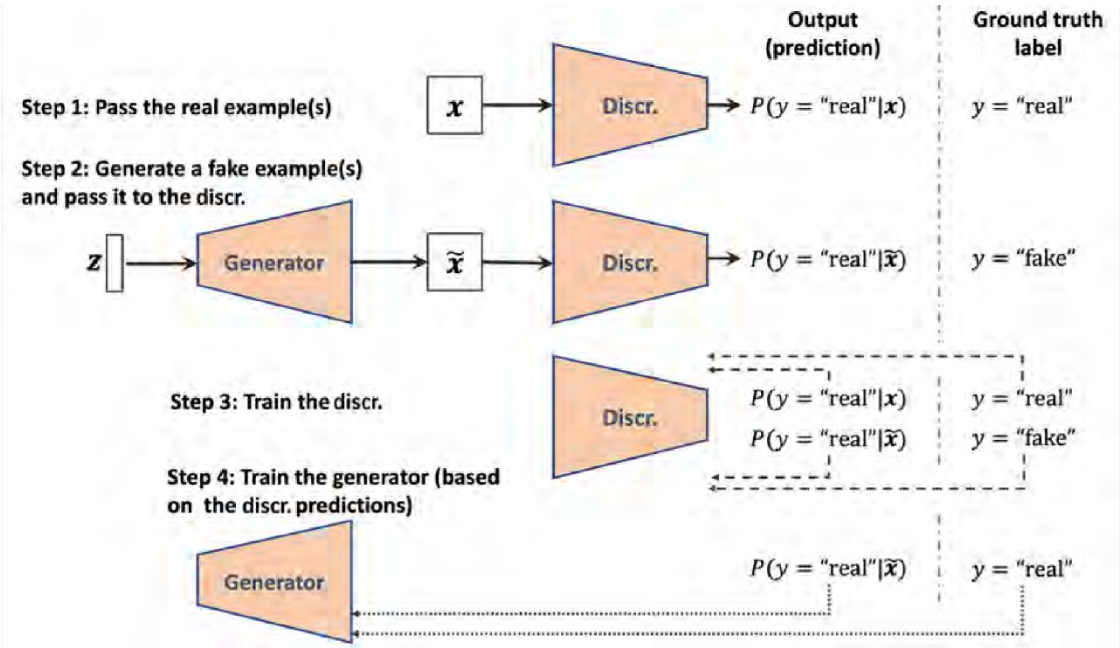
GAN modellerini eğitmek için genel optimizasyon prosedürünü ele aldığımıza göre, şimdi GAN'ları eğitirken kullanabileceğimiz çeşitli veri etiketlerini inceleyelim. Ayırıcının ikili bir sınıflandırıcı olduğu göz önüne alındığında (sınıf etiketleri sahte ve gerçek görüntüler için sırasıyla 0 ve 1'dir), ikili çapraz entropi kayıp fonksiyonunu kullanabiliriz. Bu nedenle, ayırıcı kaybı için temel gerçek etiketlerini aşağıdaki gibi belirleyebiliriz:

Ayırıcı için temel gerçek etiketleri $= \begin{cases} 1: \text{gerçek görüntüler için, yani } xx \\ 0: G \text{ çıkışları için, yani } G(z) \end{cases}$

Üreteci eğitmek için etiketler ne olacak? Üretecin gerçekçi görüntüler sentezlemesini istediğimizden, çıktıları ayırıcı tarafından gerçek olarak sınıflandırılmadığında üreteci cezalandırmak istiyoruz. Bu, üretecin kayıp fonksiyonunu hesaplarken üretecin çıktıları için temel gerçek etiketlerinin 1 olduğunu varsayacağımız anlamına gelir.

 olimpikbarbell@gmail.com

Tüm bunları bir araya getirdiğimizde, aşağıdaki şekil basit bir GAN modelindeki bireysel adımları göstermektedir:



Şekil 17.4: Bir GAN modeli oluşturmanın adımları

Bir sonraki bölümde, yeni el yazısı rakamlar üretmek için sıfırdan bir GAN uygulayacağız.

Sıfırdan bir GAN uygulamak

Bu bölümde, MNIST rakamları gibi yeni görüntüler oluşturmak için bir GAN modelinin nasıl uygulanacağını ve eğitileceğini ele alacağız. Normal bir **merkezi işlem biriminde** (**CPU**) eğitim uzun sürebileceğinden, aşağıdaki alt bölümde, hesaplamaları **grafik işlem birimlerinde** (**GPU**'lar) çalıştırmamıza olanak tanıyan Google Colab ortamının nasıl kurulacağını ele alacağız.

Google Colab üzerinde GAN modellerinin eğitimi

Bu bölümdeki kod örneklerinden bazıları, geleneksel bir dizüstü bilgisayarın veya GPU'su olmayan bir iş istasyonunun ötesine geçen kapsamlı hesaplama kaynakları gerektirebilir. CUDA ve cuDNN kütüphanelerinin yüklü olduğu NVIDIA GPU destekli bir bilgisayarınız varsa, hesaplamaları hızlandırmak için bunu kullanabilirsiniz.

Ancak, çoğumuzun yüksek performanslı bilgi işlem kaynaklarına erişimi olmadığından, ücretsiz bir bulut bilgi işlem hizmeti olan (çoğu ülkede mevcut) Google Colaboratory ortamını (genellikle Google Colab olarak anılır) kullanacağız.

Google Colab, bulut üzerinde çalışan Jupyter Notebook örnekleri sağlar; notebooklar Google Drive veya GitHub'a kaydedilebilir. Platform CPU'lar, GPU'lar ve hatta **tensör işleme birimleri** (**TPU'lar**) gibi çeşitli farklı bilgi **işlem** kaynakları sağlarken, yürütme süresinin şu anda 12 saatle sınırlı olduğunu vurgulamak önemlidir. Bu nedenle, 12 saatten daha uzun süre çalışan herhangi bir notebook kesintiye uğrayacaktır.

Bu bölümdeki kod blokları en fazla iki ila üç saatlik bir hesaplama süresine ihtiyaç duyacaktır, bu nedenle bu bir sorun olmayacaktır. Ancak, 12 saatten uzun süren diğer projeler için Google Colab kullanmaya karar verirseniz, kontrol noktası kullandığınızdan ve ara kontrol noktalarını kaydettiğinizden emin olun.

Jupyter Notebook

Jupyter Notebook, kodu etkileşimli olarak çalıştırmak ve metin belgeleri ve şekillerle birleştirmek için kullanılan bir grafik kullanıcı arayüzüdür (GUI). Çok yönlülüğü ve kullanım kolaylığı nedeniyle veri biliminde en popüler araçlardan biri haline gelmiştir.

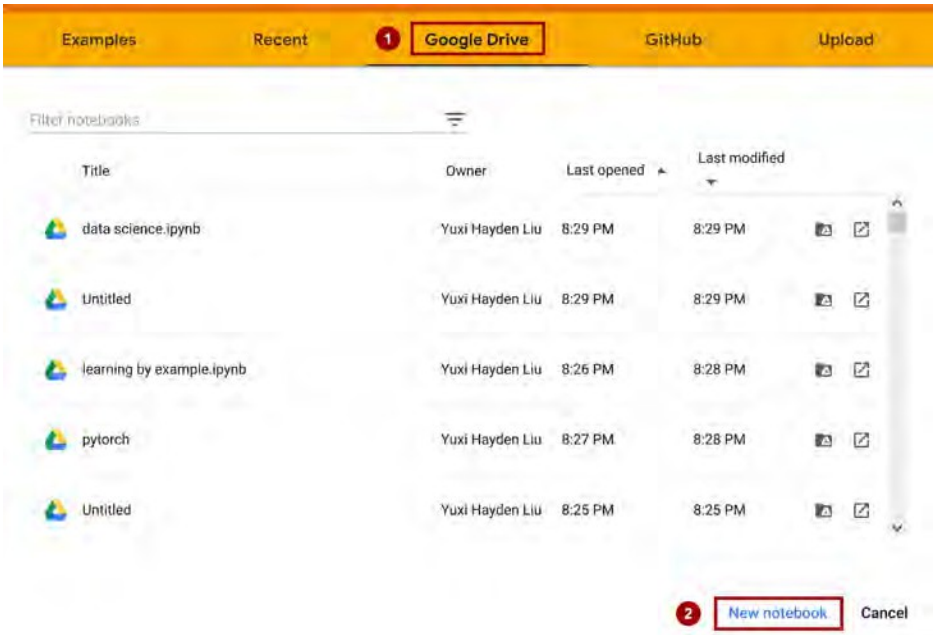
Genel Jupyter Notebook GUI'si hakkında daha fazla bilgi için lütfen https://jupyter-notebook.readthedocs.io/en/stable/ adresindeki resmi belgelere göz atın. Bu kitaptaki tüm kodlar Jupyter not defterleri şeklinde de mevcuttur ve ilk bölümün kod dizininde kısa bir tanıtım bulunabilir.

Son olarak, *Adam Rule* ve arkadaşlarının *Jupyter Notebook*'u bilimsel araştırma projelerinde etkin bir şekilde kullanmaya ilişkin Jupyter Notebook*'larda hesaplamalı analizler yazmak ve paylaşmak için on basit kural başlıklı* makalesini şiddetle tavsiye ediyoruz. https://journals.plos.org/ ploscompbiol/article?id=10.1371/journal.pcbi.1007007 adresinden ücretsiz olarak erişilebilir.

Google Colab'a erişmek çok basittir. Sizi otomatik olarak mevcut Jupyter not defterlerinizi görebileceğiniz bir istem penceresine götüren https://colab.research.google.com adresini ziyaret edebilirsiniz. Bu istem penceresinden, *Şekil 17.5*'te gösterildiği gibi **Google Drive** sekmesine tıklayın. Burası not defterini Google Drive'ınıza kaydedeceğiniz yerdir.

ilinpikbarbell@gmail.com

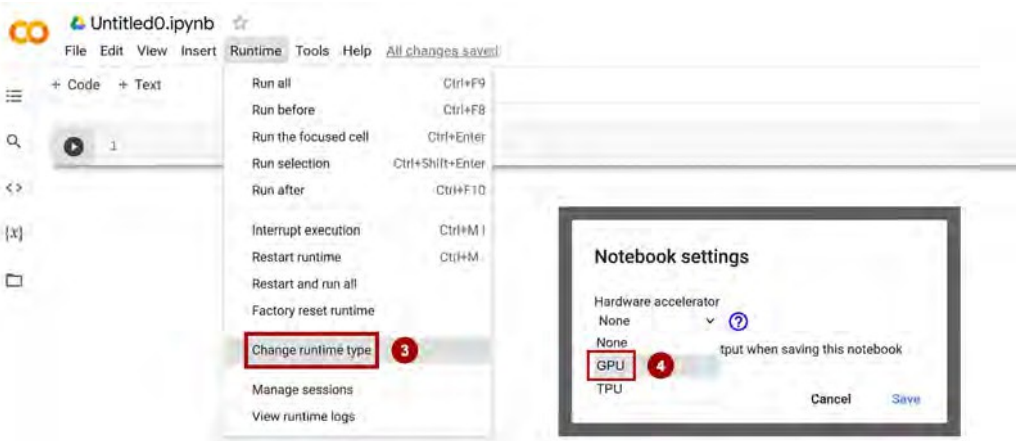
Ardından, yeni bir not defteri oluşturmak için istem penceresinin altındaki **Yeni not defteri** bağlantısına tıklayın:



Şekil 17.5: Google Colab'da yeni bir Python not defteri oluşturma

Bu sizin için yeni bir not defteri oluşturacak ve açacaktır. Bu not defterine yazdığınız tüm kod örnekleri otomatik olarak kaydedilir ve daha sonra not defterine Google Drive'ınızdan **Colab Notebooks** adlı bir dizinden erişebilirsiniz.

Bir sonraki adımda, bu not defterindeki kod örneklerini çalıştırmak için GPU'ları kullanmak istiyoruz. Bunu yapmak için, bu not defterinin menü çubuğundaki Çalışma **Zamanı** seçeneğinden **Çalışma zamanı türünü değiştir**'e tıklayın ve *Şekil 17.6*'da gösterildiği gibi **GPU'yu** seçin:



Şekil 17.6: Google Colab'da GPU'lardan yararlanma

Son adımda, bu bölüm için ihtiyaç duyacağımız Python paketlerini yüklememiz gerekiyor. Colab Notebooks ortamı zaten NumPy, SciPy ve PyTorch'un en son kararlı sürümü gibi belirli paketlerle birlikte gelir. Yazım sırasında, Google Colab'daki en son kararlı sürüm PyTorch 1.9'dur.

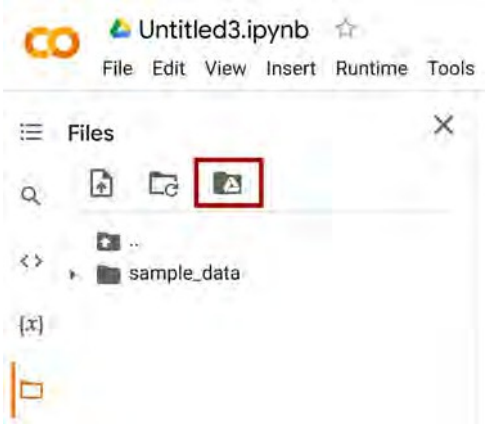
Şimdi, aşağıdaki kodu kullanarak kurulumu test edebilir ve GPU'nun kullanılabilir olduğunu doğrulayabiliriz:

```
>> import torch
>> print(torch. version )
1.9.0+cu111
>> print("GPU A v a i l a b l e :",
torch.cuda.is_available()) GPU Available: True
>> if torch.cuda.is_available():
...         cihaz = torch.device("cuda:0")
...başka:
...         aygıt = "cpu"
>> print(cihaz)
cuda:0
```

Ayrıca, modeli kişisel Google Drive'ınıza kaydetmek veya diğer dosyaları aktarmak ya da yüklemek istiyorsanız, Google Drive'ı bağlamanız gerekir. Bunu yapmak için, not defterinin yeni bir hücresinde aşağıdakileri uygulayın:

```
>> from google.colab import drive
>> drive.mount('/content/drive/')
```

Bu, Google Drive'ınıza erişen Colab Notebook'un kimliğini doğrulamak için bir bağlantı sağlayacaktır. Kimlik doğrulama talimatlarını izledikten sonra, az önce çalıştırdığınız hücrenin altındaki belirlenmiş giriş alanına kopyalayıp yapıştırmanız gereken bir kimlik doğrulama kodu sağlayacaktır. Ardından, Google Drive'ınız bağlanacak ve /content/drive/My Drive adresinden erişilebilir olacaktır. Alternatif olarak, *Şekil 17.7*'de vurgulandığı gibi GUI arayüzü aracılığıyla da bağlayabilirsiniz:

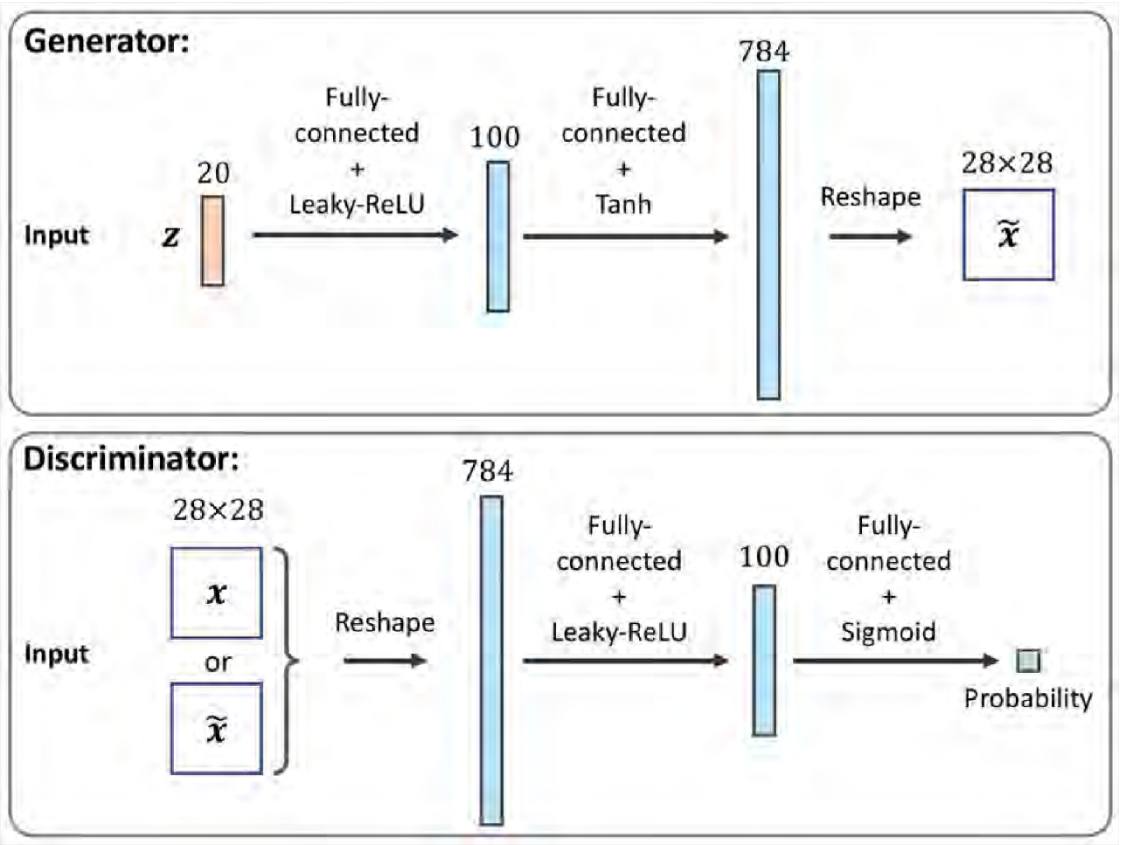


Şekil 17.7: Google Drive'ınızı bağlama

olimpikbarbell@gmail.com

Üreteç ve ayırıcı ağların uygulanması

İlk GAN modelimizin uygulamasına, *Şekil 17.8'de* gösterildiği gibi, bir veya daha fazla gizli katmana sahip iki tam bağlı ağ olarak bir üreteç ve bir ayırıcı ile başlayacağız:



Şekil 17.8: İki tam bağlı ağ olarak bir üreteç ve ayırıcıya sahip bir GAN modeli

Şekil 17.8, vanilya GAN olarak adlandıracağımız tam bağlı katmanlara dayalı orijinal *GAN'ı göstermektedir*.

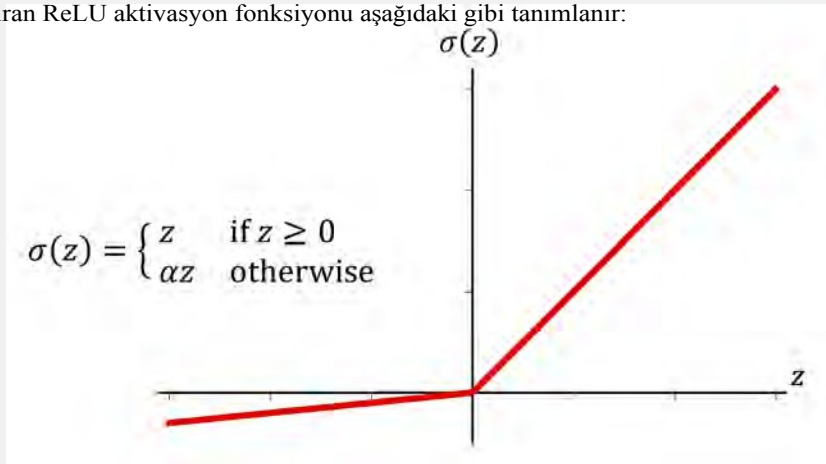
Bu modelde, her gizli katman için sızdıran ReLU aktivasyon fonksiyonunu uygulayacağız. ReLU kullanımı seyrek gradyanlarla sonuçlanır, bu da tüm girdi değerleri aralığı için gradyanlara sahip olmak istediğimizde uygun olmayabilir. Ayırıcı ağda, her gizli katmanı ayrıca bir bırakma katmanı takip eder. Ayrıca, üreteçteki çıkış katmanı hiperbolik tanjant (tanh) aktivasyon fonksiyonunu kullanır. (Öğrenmeye yardımcı olduğu için üreteç ağı için tanh aktivasyonunun kullanılması önerilir).

Diskriminatördeki çıkış katmanında logitleri elde etmek için aktivasyon fonksiyonu (yani doğrusal aktivasyon) yoktur. Alternatif olarak, olasılıkları çıktı olarak almak için sigmoid aktivasyon fonksiyonunu kullanabiliriz.

Sızdıran düzeltilmiş doğrusal birim (ReLU) aktivasyon fonksiyonu

PyTorch ile Sinir Ağı Eğitimini Paralelleştirme başlıklı *12. Bölümde,* bir NN modelinde kullanılabilecek farklı doğrusal olmayan aktivasyon fonksiyonlarını ele almıştık. Hatırlarsanız, ReLU aktivasyon fonksiyonu negatif (aktivasyon öncesi) girdileri bastıran $\sigma(z) = \max(0, z)$ olarak tanımlanmıştı; yani negatif girdiler sıfıra ayarlanır. Sonuç olarak, ReLU aktivasyon fonksiyonunun kullanılması geri yayılım sırasında seyrek gradyanlara neden olabilir. Seyrek gradiyentler her zaman zararlı değildir ve hatta sınıflandırma için modellere fayda sağlayabilir. Bununla birlikte, GAN'lar gibi bazı uygulamalarda, tüm girdi değerleri aralığı için gradyanları elde etmek faydalı olabilir; bunu, ReLU işlevinde negatif girdiler için küçük değerler üretecek şekilde küçük bir değişiklik yaparak başarabiliriz. ReLU fonksiyonunun bu değiştirilmiş versiyonu **sızdıran ReLU** olarak da bilinir. Kısacası, sızdıran ReLU aktivasyon fonksiyonu negatif girdiler için de sıfır olmayan gradyanlara izin verir ve sonuç olarak ağları genel olarak daha etkileyici hale getirir.

Sızdıran ReLU aktivasyon fonksiyonu aşağıdaki gibi tanımlanır:



Şekil 17.9: Sızdıran ReLU aktivasyon fonksiyonu

Burada, α negatif (ön aktivasyon) girişler için eğimi belirler.

 ilinpikbarbell@gmail.com

İki ağın her biri için iki yardımcı fonksiyon tanımlayacağız, PyTorch'tan bir model örnekleyeceğiz nn.Sequential sınıfını oluşturun ve katmanları açıklandığı gibi ekleyin. Kod aşağıdaki gibidir:

```
>> import torch.nn as nn
>> import numpy as np
>> import matplotlib.pyplot as plt
>> ## jeneratör için bir fonksiyon tanımlayın:
>> def make_generator_network(
...             input_size=20,
...             num_hidden_layers=1,
...             num_hidden_units=100,
...             num_output_units=784):
...         model = nn.Sequential()
...         for i  in range(num_hidden_layers):
...             model.add_module(f'fc_g{i}',
...                                 nn.Linear(input_size, num_hidden_units))
...             model.add_module(f'relu_g{i}', nn.LeakyReLU())
...             input_size = num_hidden_units
...         model.add_module(f'fc_g{num_hidden_layers}',
...                             nn.Linear(input_size, num_output_units))
...         model.add_module('tanh_g', nn.Tanh())
...         dönüş modeli
>>>
>> ## ayırıcı için bir fonksiyon tanımlayın:
>> def make_discriminator_network(
...             input_size,
...             num_hidden_layers=1,
...             num_hidden_units=100,
...             num_output_units=1):
...         model = nn.Sequential()
...         for i  in range(num_hidden_layers):
...             model.add_module(
...                 f'fc_d{i}',
...                 nn.Linear(input_size, num_hidden_units, bias=False)
...             )
...             model.add_module(f'relu_d{i}', nn.LeakyReLU())
...             model.add_module('dropout', nn.Dropout(p=0.5))
...             input_size = num_hidden_units
...         model.add_module(f'fc_d{num_hidden_layers}',
...                             nn.Linear(input_size, num_output_units))
...         model.add_module('sigmoid', nn.Sigmoid())
...         dönüş modeli
```

Daha sonra, model için eğitim ayarlarını belirleyeceğiz. Önceki bölümlerden hatırlayacağınız gibi, MNIST veri kümesindeki görüntü boyutu 28×28 pikseldir. (Bu yalnızca bir renk kanalıdır çünkü MNIST yalnızca gri tonlamalı görüntüler içerir). Giriş vektörü z'nin boyutunu da 20 olarak belirleyeceğiz. Yalnızca örnekleme amacıyla çok basit bir GAN modeli uyguladığımız ve tam bağlı katmanlar kullandığımız için, her ağda yalnızca 100 birimlik tek bir gizli katman kullanacağız. Aşağıdaki kodda, iki ağı belirleyip başlatacağız ve özet bilgilerini yazdıracağız:

```
>> image_size = (28, 28)
>> z_size = 20
>> gen_hidden_layers = 1
>> gen_hidden_size = 100
>> disc_hidden_layers = 1
>> disc_hidden_size = 100
>>> torch.manual_seed(1)
>>> gen_model = make_generator_network(
...         input_size=z_size,
...         num_hidden_layers=gen_hidden_layers,
...         num_hidden_units=gen_hidden_size,
...         num_output_units=np.prod(image_size)
... )
>> print(gen_model) Sıralı(
  (fc_g0): Linear(in_features=20, out_features=100, bias=False)
  (relu_g0): LeakyReLU(negative_slope=0.01)
  (fc_g1): Linear(in_features=100, out_features=784, bias=True)
  (tanh_g): Tanh()
)

>> disk_model = make_discriminator_network(
...         input_size=np.prod(image_size),
...         num_hidden_layers=disc_hidden_layers,
...         num_hidden_units=disc_hidden_size
... )
>> print(disk_modeli) Sıralı(
  (fc_d0): Linear(in_features=784, out_features=100, bias=False)
  (relu_d0): LeakyReLU(negative_slope=0.01)
  (dropout): Dropout(p=0.5, inplace=False)
  (fc_d1): Linear(in_features=100, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

 olinpikbarbell@gmail.com

Eğitim veri setinin tanımlanması

Bir sonraki adımda, PyTorch'tan MNIST veri kümesini yükleyeceğiz ve gerekli ön işleme adımlarını uygulayacağız. Üretecin çıkış katmanı tanh aktivasyon fonksiyonunu kullandığından, sentezlenen görüntülerin piksel değerleri (-1, 1) aralığında olacaktır. Ancak, MNIST görüntülerinin giriş pikselleri [0, 255] aralığındadır (PIL.Image.Image veri tipiyle). Bu nedenle, ön işleme adımlarında, giriş görüntü tensörlerini bir tensöre dönüştürmek için torchvision.transforms.ToTensor işlevini kullanacağız. Sonuç olarak, veri türünü değiştirmenin yanı sıra, bu işlevi çağırmak giriş piksel yoğunlukları aralığını da [0, 1] olarak değiştirecektir. Ardından, bunları -0,5 kaydırabilir ve piksel yoğunlukları [-1, 1] aralığında olacak şekilde yeniden ölçeklendirilecek şekilde 0,5 faktörü ile ölçeklendirebiliriz, bu da gradyan iniş tabanlı

```
>> import torchvision
>> from torchvision import transforms
>> image_path = './'
>> transform = transforms.Compose([
...         transforms.ToTensor(),
...         transforms.Normalize(mean=(0.5), std=(0.5)),
... ])
>> mnist_dataset = torchvision.datasets.MNIST(
...         root=image_path, train=True,
...         transform=transform, download=False
... )
>> örnek, etiket = next(iter(mnist_dataset))
>> print(f'Min: {example.min()} Max: {example.max()}')
>> print(örnek.şekil) Min: -
1.0 Maks: 1.0
torch.Size([1, 28, 28])
```

öğrenmeyi geliştirebilir:

Ayrıca, istenen rastgele dağılıma (bu kod örneğinde, en yaygın seçimler olan tekdüze veya normal) dayalı olarak rastgele bir vektör, ***z,*** oluşturacağız:

```
>> def create_noise(batch_size, z_size, mode_z):
...         if mode_z == 'uniform':
...                 input_z = torch.rand(batch_size, z_size)*2 - 1
...         elif mode_z == 'normal':
...                 input_z = torch.randn(batch_size, z_size)
...         return input_z
```

Oluşturduğumuz veri kümesi nesnesini inceleyelim. Aşağıdaki kodda, bir grup örnek alacağız ve bu örnek giriş vektörlerinin ve görüntülerinin dizi şekillerini yazdıracağız. Ayrıca, GAN modelimizin genel veri akışını anlamak için, aşağıdaki kodda, jeneratörümüz ve diskriminatörümüz için bir ileri geçiş işleyeceğiz.

İlk olarak, girdi, *z*, vektörleri grubunu üretece besleyeceğiz ve çıktısını, g_output, alacağız. Bu, sahte örnekler yığını olacak ve sahte örnekler yığınının olasılıklarını elde etmek için ayırıcı modele beslenecektir, d_proba_fake. Ayrıca, veri kümesi nesnesinden elde ettiğimiz işlenmiş görüntüler, gerçek örnekler için olasılıklar, d_proba_real ile sonuçlanacak olan ayırt edici modele beslenecektir. Kod aşağıdaki gibidir:

```
>> from torch.utils.data import DataLoader
>>> batch_size = 32
>> dataloader = DataLoader(mnist_dataset, batch_size, shuffle=False)
>> input_real, label = next(iter(dataloader))
>> input_real = input_real.view(batch_size, -1)
>>> torch.manual_seed(1)
>> mode_z = 'uniform' # ' uniform' vs. ' normal'
>>> input_z = create_noise(batch_size, z_size, mode_z)
>> print('input-z -- shape:', input_z.shape)
>> print('input-real -- shape:', input_real.shape)
input-z -- shape: torch.Size([32, 20])
input-real -- shape: torch.Size([32, 784])

>> g_çıktı = gen_model(girdi_z)
>> print('G çıktısı -- şekil:', g_output.shape) G çıktısı --
şekil: torch.Size([32, 784])

>> d_proba_real = disc_model(input_real)
>> d_proba_fake = disc_model(g_output)
>> print('Disk. (gerçek) -- şekil:', d_proba_real.shape)
>> print('Disk. (sahte) -- şekil:', d_proba_fake.shape)
Disk. (gerçek) -- şekil: torch.Size([32, 1])
Disk. (sahte) -- şekil: torch.Size([32, 1])
```

İki olasılık, d_proba_fake ve d_proba_real, modeli eğitmek için kayıp fonksiyonlarını hesaplamak için kullanılacaktır.

GAN modelinin eğitimi

Bir sonraki adım olarak, kayıp fonksiyonumuz olarak bir nn.BCELoss örneği oluşturacağız ve az önce işlediğimiz gruplarla ilişkili üreteç ve ayırıcı için ikili çapraz entropi kaybını hesaplamak için bunu kullanacağız. Bunu yapmak için, her çıktı için temel gerçek etiketlerine de ihtiyacımız var. Oluşturucu için, oluşturulan görüntüler için tahmin edilen olasılıkları içeren vektörle aynı şekle sahip 1'lerden oluşan bir vektör oluşturacağız, d_proba_fake. Ayırıcı kaybı için iki terimimiz var: d_proba_fake'i içeren sahte örnekleri tespit etme kaybı ve d_proba_real'e dayalı gerçek örnekleri tespit etme kaybı.

 ilinpikbarbell@gmail.com

Sahte terim için temel doğruluk etiketleri, torch.zeros() (veya torch.zeros_like()) fonksiyonu aracılığıyla oluşturabileceğimiz 0'lardan oluşan bir vektör olacaktır. Benzer şekilde, gerçek görüntüler için temel doğruluk değerlerini torch.ones() (veya torch.ones_like()) fonksiyonu

```
>> kayıp_fn = nn.BCELoss()
>> ## Jeneratör için kayıp
>> g_labels_real = torch.ones_like(d_proba_fake)
>>> g_loss = loss_fn(d_proba_fake, g_labels_real)
>> print(f'Jeneratör Kaybı: {g_loss:.4f}')
Jeneratör Kaybı: 0.6863

>> ## Ayırıcı için Kayıp
>> d_labels_real = torch.ones_like(d_proba_real)
>> d_labels_fake = torch.zeros_like(d_proba_fake)
>> d_loss_real = loss_fn(d_proba_real, d_labels_real)
>> d_loss_fake = loss_fn(d_proba_fake, d_labels_fake)
>> print(f'Ayırıcı Kayıpları: Gerçek {d_loss_real:.4f} Sahte {d_loss_
fake:.4f}')
Ayırıcı Kayıpları: Gerçek 0,6226 Sahte 0,7007
```

aracılığıyla oluşturabiliriz, bu da 1'lerden oluşan bir vektör oluşturur:

Önceki kod örneği, bir GAN modelinin eğitilmesinin arkasındaki genel konsepti anlamak amacıyla farklı kayıp terimlerinin adım adım hesaplanmasını göstermektedir. Aşağıdaki kod GAN modelini kuracak ve bu hesaplamaları bir for döngüsüne dahil edeceğimiz eğitim döngüsünü uygulayacaktır.

Gerçek veri kümesi için veri yükleyiciyi, üretici ve ayırıcı modelin yanı sıra iki modelin her biri için ayrı bir Adam optimize ediciyi kurarak başlayacağız:

```
>>> batch_size = 64
>>> torch.manual_seed(1)
>> np.random.seed(1)
>> mnist_dl = DataLoader(mnist_dataset, batch_size=batch_size,
...                      shuffle=True, drop_last=True)

>>> gen_model = make_generator_network(
...     input_size=z_size,
...     num_hidden_layers=gen_hidden_layers,
...     num_hidden_units=gen_hidden_size,
...     num_output_units=np.prod(image_size)
... ).to(cihaz)
>> disk_model = make_discriminator_network(
...     input_size=np.prod(image_size),
...     num_hidden_layers=disc_hidden_layers,
...     num_hidden_units=disc_hidden_size
... ).to(cihaz)
```

```
>> kayıp_fn = nn.BCELoss()
>> g_optimizer = torch.optim.Adam(gen_model.parameters())
>> d_optimizer = torch.optim.Adam(disc_model.parameters())
```

Buna ek olarak, model ağırlıklarına göre kayıp gradyanlarını hesaplayacağız ve iki ayrı Adam optimize edici kullanarak üreteç ve ayrıştırıcı parametrelerini optimize edeceğiz. Ayırıcı ve üreteci eğitmek için aşağıdaki gibi iki fayda fonksiyonu yazacağız:

```
>> ## Ayırıcıyı eğitin
>> def d_train(x):
...     disc_model.zero_grad()
...     # Ayırıcıyı gerçek bir parti ile eğitin batch_size =
...     x.size(0)
...     x = x.view(batch_size, -1).to(device)
...     d_labels_real = torch.ones(batch_size, 1, device=device)
...     d_proba_real = disc_model(x)
...     d_loss_real = loss_fn(d_proba_real, d_labels_real) #
...     Ayırıcıyı sahte bir yığın üzerinde eğitin
...     input_z = create_noise(batch_size, z_size, mode_z).to(device)
...     g_output = gen_model(input_z)
...     d_proba_fake = disc_model(g_output)
...     d_labels_fake = torch.zeros(batch_size, 1, device=device)
...     d_loss_fake = loss_fn(d_proba_fake, d_labels_fake)
...     # gradyan geri yayılımı ve SADECE D'nin parametrelerini
...     optimize edin d_loss = d_loss_real + d_loss_fake
...     d_loss.backward()
...     d_optimizer.step()
...     return d_loss.data.item(), d_proba_real.detach(), \
...             d_proba_fake.detach()
>>>
>> ## Jeneratörü eğitin
>> def g_train(x):
...     gen_model.zero_grad()
...     batch_size = x.size(0)
...     input_z = create_noise(batch_size, z_size, mode_z).to(device)
...     g_labels_real = torch.ones(batch_size, 1, device=device)
...
...     g_çıktı = gen_model(girdi_z)
...     d_proba_fake = disc_model(g_output)
...     g_loss = loss_fn(d_proba_fake, g_labels_real)
```

alinpikbarbell@gmail.com

```
...     # gradient backprop ve SADECE G'nin parametrelerini
        optimize edin
...     g_loss.backward()
...     g_optimizer.step() return
...     g_loss.data.item()
```

Daha sonra, 100 epok boyunca üretecin ve ayrıştırıcının eğitimi arasında geçiş yapacağız. Her bir epok için sırasıyla üretecin kaybını, ayırt edicinin kaybını ve gerçek veri ile sahte verinin kaybını kaydedeceğiz. Ayrıca, her epoktan sonra, create_samples() fonksiyonunu çağırarak mevcut üreteç modelini kullanarak sabit bir gürültü girişinden bazı örnekler üreteceğiz. Sentezlenen görüntüleri bir Python listesinde saklayacağız. Kod aşağıdaki gibidir:

```
>>> fixed_z = create_noise(batch_size, z_size, mode_z).to(device)
>> def create_samples(g_model, input_z):
...     g_çıktı = g_model(girdi_z)
...     images = torch.reshape(g_output, (batch_size, *image_size))
...     return (images+1)/2.0
>>>
>> epoch_samples = []
>> all_d_losses = []
>> all_g_losses = []
>> all_d_real = []
>> all_d_fake = []
>> num_epochs = 100
>>>
>> for epoch in range(1, num_epochs+1):
...     d_losses, g_losses = [], []
...     d_vals_real, d_vals_fake = [], []
...     for i, (x, _) in enumerate(mnist_dl):
...         d_loss, d_proba_real, d_proba_fake = d_train(x)
...         d_losses.append(d_loss)
...         g_losses.append(g_train(x))
...         d_vals_real.append(d_proba_real.mean().cpu())
...         d_vals_fake.append(d_proba_fake.mean().cpu())
...
...     all_d_losses.append(torch.tensor(d_losses).mean())
...     all_g_losses.append(torch.tensor(g_losses).mean())
...     all_d_real.append(torch.tensor(d_vals_real).mean())
...     all_d_fake.append(torch.tensor(d_vals_fake).mean())
...     print(f'Epoch {epoch:03d} | Ortalama Kayıplar >>'
...             f' G/D {all_g_losses[-1]:.4f}/{all_d_losses[-1]:.4f}'
...             f' [D-Real: {all_d_real[-1]:.4f}'
...             f' D-Sahte: {all_d_fake[-1]:.4f}]')
```

```
...         epoch_samples.append(
...                 create_samples(gen_model, fixed_z).detach().cpu().numpy()
...         )

Dönem 001 | Ortalama Kayıplar >> G/D 0,9546/0,8957 [D-Gerçek: 0 , 8074 D-Sahte:
0 , 4687]
Dönem 002 | Ortalama Kayıplar >> G/D 0,9571/1,0841 [D-Gerçek: 0,6346 D-
Sahte: 0,4155] Dönem …
Epoch 100 | Ortalama Kayıplar >> G/D 0,8622/1,2878 [D-Real: 0,5488 D-Fake:
0 , 4518]
```

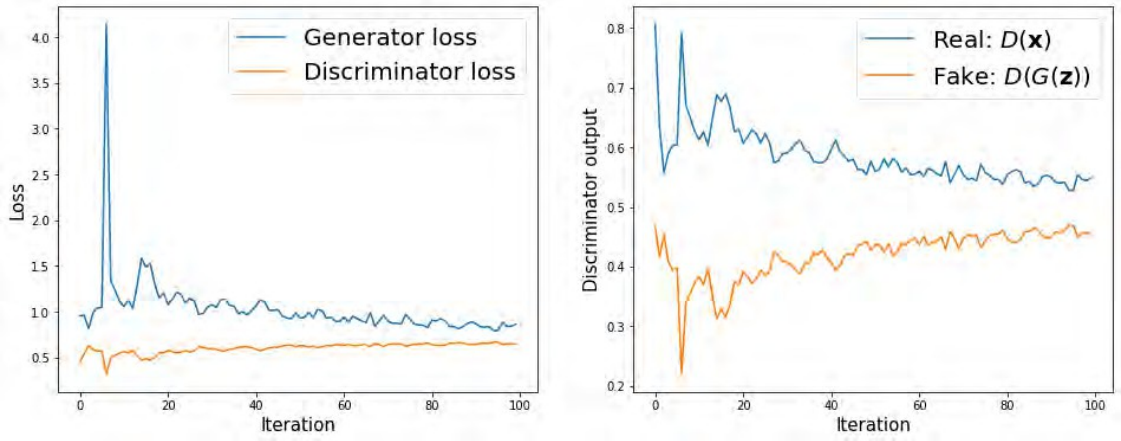
Google Colab üzerinde bir GPU kullanarak, önceki kod bloğunda uyguladığımız eğitim süreci bir saatten daha kısa bir sürede tamamlanmalıdır. (Yeni ve yetenekli bir CPU ve GPU'ya sahipseniz kişisel bilgisayarınızda daha da hızlı olabilir). Model eğitimi tamamlandıktan sonra, her iki alt ağın davranışını analiz etmek ve yakınsayıp yakınsamadıklarını değerlendirmek için ayırıcı ve jeneratör kayıplarını çizmek genellikle yararlıdır.

Her iterasyonda ayırıcı tarafından hesaplanan gerçek ve sahte örnek gruplarının ortalama olasılıklarını çizmek de faydalıdır. Bu olasılıkların 0,5 civarında olmasını bekliyoruz, bu da ayırıcının gerçek ve sahte görüntüleri güvenle ayırt edemediği anlamına geliyor:

```
>> import itertools
>>> fig = plt.figure(figsize=(16, 6))
>> ## Kayıpların çizilmesi
>> ax = fig.add_subplot(1, 2, 1)
>> plt.plot(all_g_losses, label='Jeneratör kaybı')
>>> half_d_losses = [all_d_losses içindeki all_d_loss için all_d_loss/2]
>> plt.plot(half_d_losses, label='Discriminator l o s s ')
>> plt.legend(fontsize=20)
>>  ax.set_xlabel('Iteration', size=15)
>>  ax.set_ylabel('Kayıp', size=15)
>>>
>> ## Ayırıcının çıktılarını çizme
>> ax = fig.add_subplot(1, 2, 2)
>> plt.plot(all_d_real, label=r'Gerçek: $D(\mathbf{x})$')
>> plt.plot(all_d_fake, label=r'Fake: $D(G(\mathbf{z}))$')
>> plt.legend(fontsize=20)
>>  ax.set_xlabel('Iteration', size=15)
>> ax.set_ylabel('Ayrıştırıcı çıktısı', size=15)
>> plt.show()
```

 ilinpikbarbell@gmail.com

Şekil 17.10 sonuçları göstermektedir:



Şekil 17.10: Ayırıcı performansı

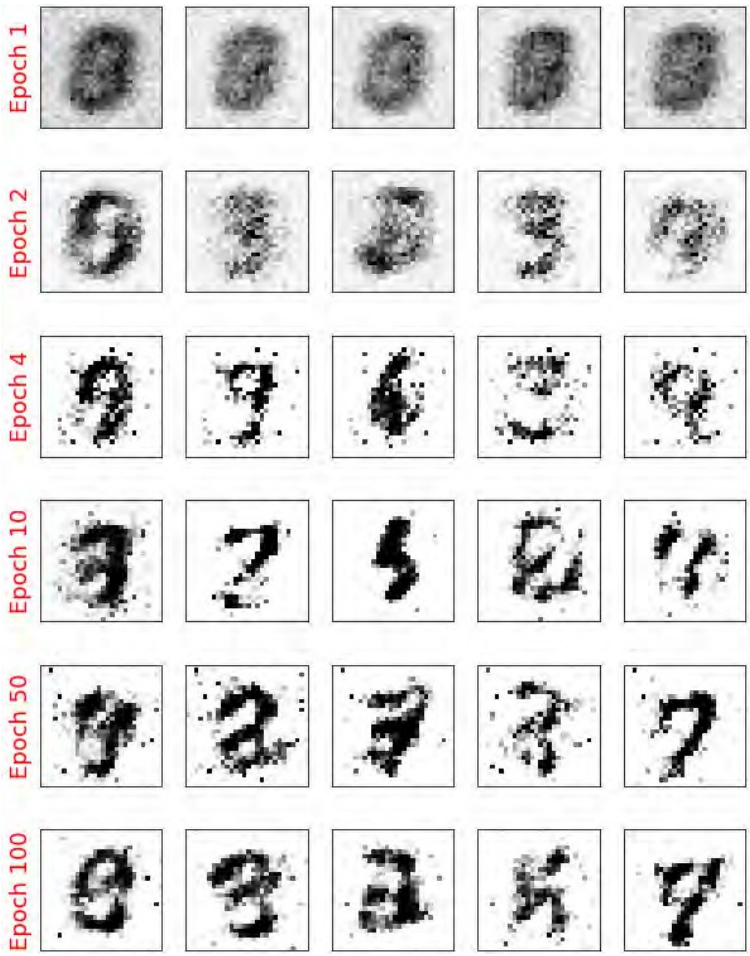
Bir önceki şekildeki ayırıcı çıktılarından da görebileceğiniz gibi, eğitimin ilk aşamalarında ayırıcı gerçek ve sahte örnekler arasında oldukça doğru bir ayrım yapmayı hızlı bir şekilde öğrenebilmiştir; yani, sahte örneklerin 0'a yakın olasılıkları ve gerçek örneklerin 1'e yakın olasılıkları vardı. Bunun nedeni, sahte örneklerin gerçek örneklere hiç benzememesiydi; bu nedenle, gerçek ve sahte arasında ayrım yapmak oldukça kolaydı. Eğitim ilerledikçe, üreteç gerçekçi görüntüler sentezleme konusunda daha iyi hale gelecek ve bu da hem gerçek hem de sahte örneklerin olasılıklarının 0,5'e yakın olmasıyla sonuçlanacaktır.

Ayrıca, üretecin çıktılarının, yani sentezlenen görüntülerin eğitim sırasında nasıl değiştiğini de görebiliriz. Aşağıdaki kodda, bir dizi epok için üreteç tarafından üretilen görüntülerden bazılarını görselleştireceğiz:

```
>> selected_epochs = [1, 2, 4, 10, 50, 100]
>>> fig = plt.figure(figsize=(10, 14))
>> for i,e in enumerate(selected_epochs):
...     for j in range(5):
...         ax = fig.add_subplot(6, 5, i*5+j+1)
...         ax.set_xticks([])
...         ax.set_yticks([])
...         j == 0 ise:
...             ax.text(
...                 -0.06, 0.5, f'Epoch {e}',
...                 rotation=90, size=18, color='red',
...                 horizontalalignment='right',
...                 verticalalignment='center',
...                 transform=ax.transAxes
...             )
```

```
...
...             görüntü = epoch_samples[e-1][j]
...             ax.imshow(image, cmap='gray_r')
...
>> plt.show()
```

Şekil 17.11 üretilen görüntüleri göstermektedir:



Şekil 17.11: Jeneratör tarafından üretilen görüntüler

Şekil 17.11'den de görebileceğiniz gibi, üretici ağ eğitim ilerledikçe giderek daha gerçekçi görüntüler üretmiştir. Bununla birlikte, 100 epoktan sonra bile, üretilen görüntüler hala MNIST veri kümesinde bulunan el yazısı rakamlardan çok farklı görünmektedir.

Bu bölümde, hem üretici hem de ayırıcı için yalnızca tek bir tam bağlı gizli katmana sahip çok basit bir GAN modeli tasarladık. GAN modelini MNIST veri kümesi üzerinde eğittikten sonra, yeni el yazısı rakamlarıyla henüz tatmin edici olmasa da umut verici sonuçlar elde etmeyi başardık.

 olimpikbarbell@gmail.com

Derin Evrişimli Sinir Ağları ile Görüntülerin Sınıflandırılması başlıklı *14. Bölümde* öğrendiğimiz gibi, evrişimli katmanlara sahip NN mimarileri, görüntü sınıflandırma söz konusu olduğunda tam bağlantılı katmanlara göre çeşitli avantajlara sahiptir. Benzer bir şekilde, görüntü verileriyle çalışmak için GAN modelimize evrişimsel katmanlar eklemek sonucu iyileştirebilir. Bir sonraki bölümde, hem üretici hem de ayırıcı ağlar için konvolüsyonel katmanlar kullanan bir **derin** konvolüsyonel **GAN** (**DCGAN**) uygulayacağız.

Konvolüsyonel ve Wasserstein GAN kullanarak sentezlenmiş görüntülerin kalitesinin iyileştirilmesi

Bu bölümde, bir önceki GAN örneğinde gördüğümüz performansı geliştirmemizi sağlayacak bir DCGAN uygulayacağız. Ek olarak, ekstra bir anahtar teknik olan **Wasserstein GAN** (**WGAN**) hakkında kısaca konuşacağız.

Bu bölümde ele alacağımız teknikler aşağıdakileri içerecektir:

- Transpoze konvolüsyon
- Toplu normalleştirme (BatchNorm)
- WGAN

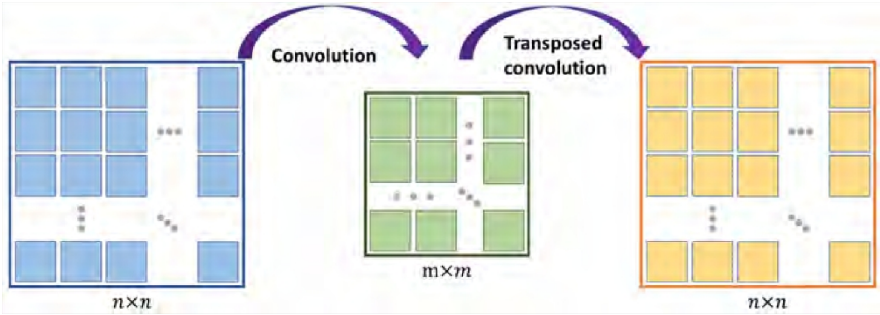
DCGAN 2016 yılında *A. Radford*, *L. Metz* ve *S. Chintala* tarafından https://arxiv.org/pdf/1511.06434.pdf adresinden ücretsiz olarak erişilebilen *Unsupervised representation learning with deep convolutional generative adversarial networks* adlı makalelerinde önerilmiştir. Bu makalede araştırmacılar, hem üretici hem de ayırıcı ağlar için konvolüsyonel katmanlar kullanmayı önermişlerdir. Rastgele bir vektör olan ***z'den*** başlayarak, DCGAN ilk olarak ***z'yi*** uygun boyutta yeni bir vektöre yansıtmak için tam bağlı bir katman kullanır, böylece çıkış görüntüsü boyutundan daha küçük olan uzamsal bir konvolüsyon temsiline ($h \times w \times c$) yeniden şekillendirilebilir. Daha sonra, özellik haritalarını istenen çıktı görüntü boyutuna yükseltmek için **transpoze konvolüsyon** olarak bilinen bir dizi **konvolüsyonel** katman kullanılır.

Transpoze konvolüsyon

*Bölüm 14'*te, bir ve iki boyutlu uzaylarda konvolüsyon işlemini öğrendiniz. Özellikle, dolgu ve adım seçimlerinin çıktı özellik haritalarını nasıl değiştirdiğini inceledik. Bir konvolüsyon işlemi genellikle özellik uzayını aşağı örneklemek için kullanılırken (örneğin, adım 2'ye ayarlanarak veya bir konvolüsyon katmanından sonra bir havuzlama katmanı eklenerek), özellik uzayını *yukarı örneklemek* için genellikle *aktarılmış* bir *konvolüsyon işlemi kullanılır.*

Transpoze konvolüsyon işlemini anlamak için basit bir düşünce deneyinden geçelim. Elimizde $n \times n$ boyutunda bir girdi özellik haritası olduğunu varsayalım. Daha sonra, bu n×n *girişe* belirli dolgu ve adım parametreleriyle bir 2B konvolüsyon işlemi uygulayarak $m \times m$ *boyutunda* bir çıkış özellik haritası elde ederiz. Şimdi soru şu: Giriş ve çıkış arasındaki bağlantı modellerini korurken bu m×m çıkış özellik haritasından başlangıç boyutu n×n *olan* bir özellik haritası elde etmek için başka bir konvolüsyon işlemini nasıl uygulayabiliriz? Gerçek matris değerlerinin değil, yalnızca n×n *giriş matrisinin* şeklinin geri kazanıldığını unutmayın.

*Şekil 17.12'*de gösterildiği gibi, aktarılmış konvolüsyonun yaptığı budur:

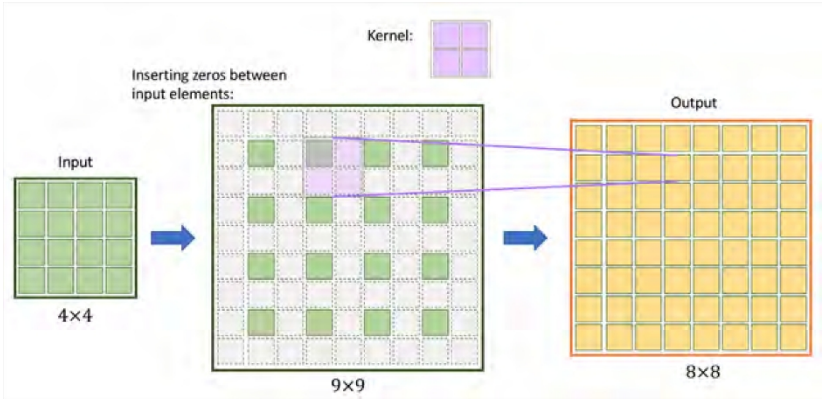


Şekil 17.12: Transpoze konvolüsyon

Transpoze konvolüsyona karşı dekonvolüsyon

Transpoze konvolüsyon aynı zamanda **kesirli konvolüsyon olarak** da adlandırılır. Derin öğrenme literatüründe, dönüştürülmüş **konvolüsyona** atıfta bulunmak için kullanılan bir diğer yaygın terim de **dekonvolüsyondur**. Bununla birlikte, dekonvolüsyonun orijinal olarak bir özellik haritası, $\boldsymbol{x}$, üzerinde ağırlık parametreleri, $\boldsymbol{w}$, ile özellik haritası üreten bir konvolüsyon işleminin, f, tersi olarak tanımlandığını unutmayın. $\boldsymbol{x}, f_{\boldsymbol{w}}(\boldsymbol{x}) = \boldsymbol{x}$. Bir dekonvolüsyon fonksiyonu, g, daha sonra $g(f(\boldsymbol{x}; \boldsymbol{w}); \boldsymbol{w}) = \boldsymbol{x}$ olarak tanımlanabilir. Bununla birlikte, dekonvolüsyonun gerçek değerleri değil, yalnızca özellik uzayının boyutluluğunu geri kazanmaya odaklandığını unutmayın.

Transpoze konvolüsyon kullanarak özellik haritalarını üst örnekleme, giriş özellik haritalarının öğeleri arasına 0'lar ekleyerek çalışır. *Şekil 17.13,* 2×2 adım ve 2×2 çekirdek boyutu ile 4×4 boyutundaki bir girdiye aktarılmış konvolüsyon uygulanmasının bir örneğini göstermektedir. Merkezdeki 9×9 boyutundaki matris, giriş özellik haritasına bu tür 0'ların eklenmesinden sonraki sonuçları göstermektedir. Ardından, 1 adımlı 2×2 çekirdeği kullanarak normal bir konvolüsyon gerçekleştirmek 8×8 boyutunda bir çıktı ile sonuçlanır. Çıktı üzerinde 2'lik bir adımla normal bir konvolüsyon gerçekleştirerek geriye doğru yönü doğrulayabiliriz, bu da orijinal girdi boyutuyla aynı olan 4×4 boyutunda bir çıktı özellik haritasıyla sonuçlanır:



Şekil 17.13: 4×4 girişe aktarılmış konvolüsyon uygulanması

 olimpikbarbell@gmail.com

Şekil 17.13'te aktarılmış konvolüsyonun genel olarak nasıl çalıştığı gösterilmektedir. Girdi boyutu, çekirdek boyutu, adımlar ve dolgu varyasyonlarının çıktıyı değiştirebileceği çeşitli durumlar vardır. Tüm bu farklı durumlar hakkında daha fazla bilgi edinmek istiyorsanız *Vincent Dumoulin* ve *Francesco Visin* tarafından hazırlanan *A Guide to Convolution Arithmetic for Deep Learning* (*Derin Öğrenme için Konvolüsyon Aritmetiği Kılavuzu*), 2018 (https://arxiv.org/pdf/1603.07285.pdf.) eğitimine bakın.

Toplu normalizasyon

BatchNorm, 2015 yılında Sergey Ioffe ve Christian Szegedy tarafından *Batch Normal- ization* makalesinde tanıtılmıştır: *Accelerating Deep Network Training by Reducing Internal Covariate Shiß adlı makalede tanıtılmıştır ve* bu makaleye arXiv üzerinden https://arxiv.org/pdf/1502.03167.pdf adresinden erişebilirsiniz. BatchNorm'un arkasındaki ana fikirlerden biri, katman girdilerini normalleştirmek ve eğitim sırasında dağılımlarındaki değişiklikleri önlemektir, bu da daha hızlı ve daha iyi yakınsama sağlar.

BatchNorm, hesaplanan istatistiklerine dayalı olarak bir mini özellik yığınını dönüştürür. Bir evrişim katmanından sonra elde edilen net ön aktivasyon özellik haritalarının $[m \times c \times h \times w]$ şeklinde dört boyutlu bir tensör olan ***Z'de*** *olduğunu* varsayalım; burada m, yığındaki örnek sayısı (yani yığın boyutu), h×w özellik haritalarının uzamsal boyutu ve c kanal sayısıdır. BatchNorm aşağıdaki gibi üç adımda özetlenebilir:

1. Her bir mini parti için net girdilerin ortalamasını ve standart sapmasını hesaplayın:

$$\mu_B = \frac{1}{m\,mh\,m\,mm} \sum_{iiiii} Z[iiiii]$$

$$\sigma_B^2 = \frac{1}{m\,mh\,m\,mm} \sum_{iiiii} (Z[iiiii] - \mu_B)^2$$

   burada μ_B ve σ_B^2 her ikisi de c boyutuna sahiptir.

2. Toplu işteki tüm örnekler için net girdileri standartlaştırın:

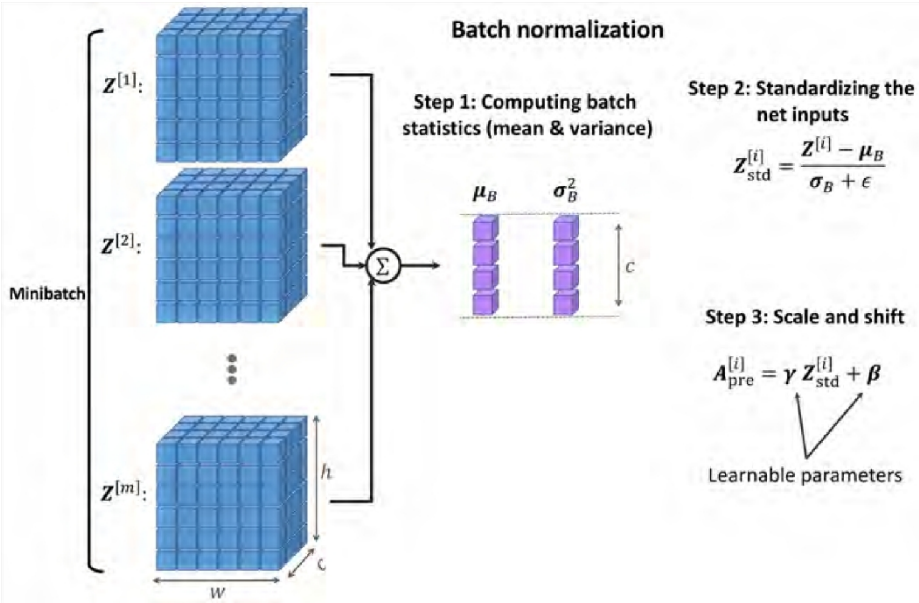
$$Z_{\text{std}}[i] = \frac{Z[i] - \mu_B}{\sigma_B + \epsilon\epsilon}$$

   Burada $\epsilon\epsilon$ sayısal kararlılık için küçük bir sayıdır (yani, sıfıra bölmekten kaçınmak için).

3. Normalleştirilmiş ağ girdilerini, c boyutunda (kanal sayısı) iki öğrenilebilir parametre vektörü, $\gamma\gamma$ ve Q, kullanarak ölçeklendirin ve kaydırın:

$$A_{\text{önceden}}[i] = \gamma\gamma Z_{\text{std}}[i] + Q$$

Şekil 17.14 süreci göstermektedir:



Şekil 17.14: Toplu normalleştirme süreci

BatchNorm'un ilk adımında, mini partinin ortalaması, μ_B , ve standart sapması, σ_B , bir araya getirilir. Hem μ_B hem de σ_B c boyutunda vektörlerdir (burada c kanal sayısıdır). Daha sonra, bu istatistikler *adım 2'*de her bir mini partideki örnekleri z-skor normalizasyonu (standardizasyon) yoluyla ölçeklendirmek için kullanılır std ve standartlaştırılmış net girdiler $Z[i]$ elde edilir. Sonuç olarak, bu net girdiler ortalama merkezlidir ve *birim varyansa* sahiptir, bu da genellikle gradyan iniş tabanlı optimizasyon için yararlı bir özelliktir. Öte yandan, net girdileri her zaman farklı mini gruplar arasında aynı özelliklere sahip olacak şekilde normalleştirmek, ki bunlar çok çeşitli olabilir, NN'lerin temsil kapasitesini ciddi şekilde etkileyebilir. Bu, $\sigma(x)$'e sigmoid aktivasyondan sonra 0'a yakın değerler için doğrusal bir bölge ile sonuçlanan xx $N(0,1)$ özelliği göz önünde bulundurularak anlaşılabilir,

Q ve $\gamma\gamma$, c boyutunda (kanal sayısı) vektörler olup BatchNorm'un normalleştirilmiş özelliklerin kaymasını ve yayılmasını kontrol etmesini sağlar.

Eğitim sırasında, çalışan ortalamalar, μ_B , ve çalışan varyans, $\sigma^2{}_B$, hesaplanır, bunlar ayarlanmış parametreler, Q ve $\gamma\gamma$ ile birlikte değerlendirmede test örneklerini normalleştirmek için kullanılır.

 olinpikbarbell@gmail.com

BatchNorm optimizasyona neden yardımcı olur?

Başlangıçta BatchNorm, eğitim sırasında güncellenen ağ parametreleri nedeniyle bir katmanın aktivasyonlarının dağılımında meydana gelen değişiklikler olarak tanımlanan **iç kovaryans kaymasını** azaltmak için geliştirilmiştir.

Bunu basit bir örnekle açıklamak gerekirse, epok 1'de ağdan geçen sabit bir yığın düşünün. Bu yığın için her katmanın aktivasyonlarını kaydediyoruz. Tüm eğitim veri kümesi boyunca yineleme yaptıktan ve model parametrelerini güncelledikten sonra, daha önce sabitlenen yığının ağdan geçtiği ikinci epoğu başlatırız. Ardından, birinci ve ikinci epoklardaki katman aktivasyonlarını karşılaştırırız. Ağ parametreleri değiştiği için aktivasyonların da değiştiğini gözlemleriz. Bu olgu, NN eğitimini yavaşlattığına inanılan **iç kovaryans kayması olarak** adlandırılır.

Ancak 2018 yılında S. Santurkar, D. Tsipras, A. Ilyas ve A. Madry, BatchNorm'u bu kadar etkili kılan şeyin ne olduğunu daha fazla araştırmıştır. Araştırmacılar çalışmalarında, BatchNorm'un iç kovaryans kayması üzerindeki etkisinin marjinal olduğunu gözlemlemişlerdir. Deneylerinin sonucuna dayanarak, BatchNorm'un etkinliğinin, bunun yerine, konveks olmayan optimizasyonu daha sağlam hale getiren kayıp fonksiyonunun daha pürüzsüz bir yüzeyine d a y a n d ı ğ ı n ı varsaymışlardır.

Bu sonuçlar hakkında daha fazla bilgi edinmek istiyorsanız, http://papers.nips.cc/paper/7515-how-does-batch-normalization-help-optimization.pdf adresinden ücretsiz olarak erişebileceğiniz Toplu Normalleştirme *Optimizasyona* Nasıl *Yardımcı* Olur*?*

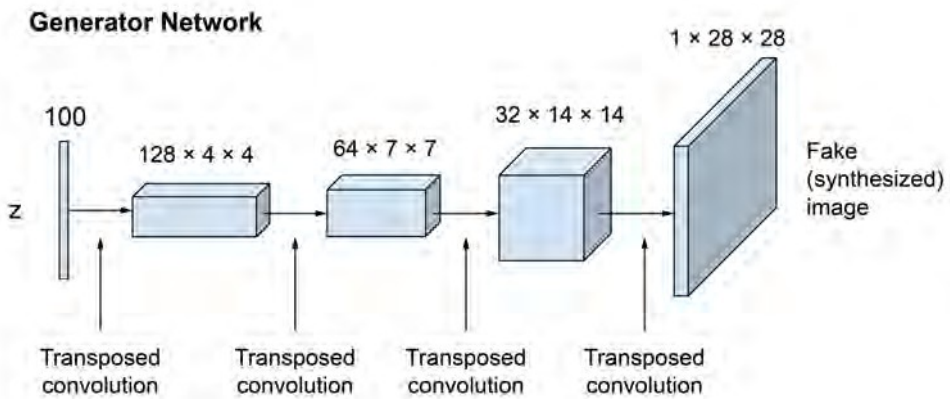
PyTorch API, modellerimizi tanımlarken bir katman olarak kullanabileceğimiz nn.BatchNorm2d() (1D girdi için nn.BatchNorm1d()) adlı bir sınıf sağlar; Batch- Norm için tanımladığımız tüm adımları gerçekleştirecektir. Öğrenilebilir parametreler olan $\gamma\gamma$ ve Q'yi güncelleme davranışının, modelin bir eğitim modeli olup olmadığına bağlı olduğunu unutmayın. Bu parametreler yalnızca eğitim sırasında öğrenilir ve daha sonra değerlendirme sırasında normalleştirme için kullanılır.

Üreteç ve ayırıcının uygulanması

Bu noktada, şimdi uygulayacağımız bir DCGAN modelinin ana bileşenlerini ele aldık. Üreteç ve ayırıcı ağların mimarileri aşağıdaki iki şekilde özetlenmiştir.

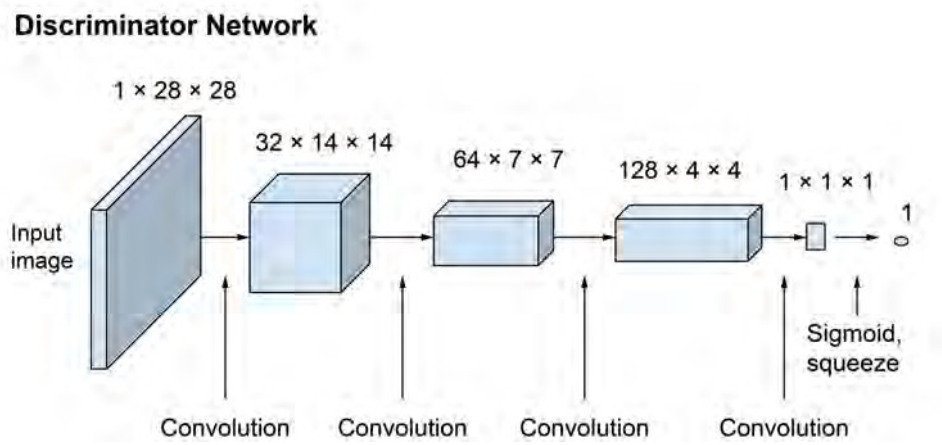
Üreteç, girdi olarak 100 boyutunda bir z vektörü alır. Ardından, nn.ConvTranspose2d() kullanılarak bir dizi dönüştürülmüş konvolüsyon, elde edilen özellik haritalarının uzamsal boyutu 28×28'e ulaşana kadar özellik haritalarını yukarı örneklemektedir. Kanal sayısı, gri tonlamalı bir görüntü oluşturmak için yalnızca bir çıkış filtresi kullanan sonuncusu hariç, her dönüştürülmüş evrişim katmanından sonra yarı yarıya azaltılır. Her bir dönüştürülmüş evrişimsel katmanı, tanh aktivasyonunu (BatchNorm olmadan) kullanan sonuncusu hariç, BatchNorm ve sızdıran ReLU aktivasyon fonksiyonları takip eder.

Üreteç mimarisi (her katmandan sonraki özellik haritaları) *Şekil 17.15*'te gösterilmektedir:



Şekil 17.15: Jeneratör ağı

Ayırıcı, dört evrişimsel katmandan geçirilen 1×28×28 boyutundaki görüntüleri alır. İlk üç evrişimsel katman, özellik haritalarının kanal sayısını artırırken uzamsal boyutluluğu 4 azaltır. Her bir evrişimsel katmanı ayrıca BatchNorm ve sızdıran ReLU aktivasyonu takip eder. Son evrişimsel katman, çıktının uzamsal boyutluluğunu 1×1×1'e düşürmek için 4×4 boyutunda çekirdekler ve tek bir filtre kullanır. Son olarak, konvolüsyonel çıktıyı bir sigmoid fonksiyonu takip eder ve tek boyuta sıkıştırılır:



Şekil 17.16: Ayırıcı ağ

 ilinpikbarbell@gmail.com

Konvolüsyonel GAN'lar için mimari tasarım hususları

Özellik haritalarının sayısının üretici ve ayırıcı arasında farklı eğilimler izlediğine dikkat edin. Üreteçte, çok sayıda özellik haritasıyla başlıyoruz ve son katmana doğru ilerledikçe bunları azaltıyoruz. Öte yandan, ayrıştırıcıda az sayıda kanalla başlıyoruz ve son katmana doğru bu sayıyı artırıyoruz. Bu, özellik haritalarının sayısı ve özellik haritalarının uzamsal boyutu ters sırada olan CNN'lerin tasarlanması için önemli bir noktadır. Özellik haritalarının uzamsal boyutu arttığında, özellik haritalarının sayısı azalır ve bunun tersi de geçerlidir.

Buna ek olarak, bir BatchNorm katmanını takip eden katmanda yanlılık birimlerinin kullanılmasının genellikle tavsiye edilmediğini unutmayın. BatchNorm zaten bir kaydırma parametresi olan Q'ye sahip olduğundan, bu durumda bias birimlerini kullanmak gereksiz olacaktır. Belirli bir katman için bias birimlerini nn.ConvTranspose2d veya nn.Conv2d içinde bias=False olarak ayarlayarak atlayabilirsiniz.

Üreteç ve ayırıcı ağ sınıfını oluşturmak için yardımcı fonksiyonun kodu aşağıdaki gibidir:

```
>> def make_generator_network(input_size, n_filters):
...     model = nn.Sequential(
...         nn.ConvTranspose2d(input_size, n_filters*4, 4,
...                            1, 0, bias=False),
...         nn.BatchNorm2d(n_filters*4),
...         nn.LeakyReLU(0.2),
...         nn.ConvTranspose2d(n_filters*4, n_filters*2,
...                            3, 2, 1, bias=False),
...         nn.BatchNorm2d(n_filters*2),
...         nn.LeakyReLU(0.2),
...         nn.ConvTranspose2d(n_filters*2, n_filters,
...                            4, 2, 1, bias=False),
...         nn.BatchNorm2d(n_filters),
...         nn.LeakyReLU(0.2),
...         nn.ConvTranspose2d(n_filters, 1, 4, 2, 1,
...                            bias=False),
...         nn.Tanh()
...     )
...     dönüş modeli
>>>
>> class Ayrıştırıcı(nn.Modül):
...     def init (self, n_filters):
...         super(). init ()
...         self.network = nn.Sequential(
...             nn.Conv2d(1, n_filters, 4, 2, 1, bias=False),
```

```
...                 nn.LeakyReLU(0.2),
...                 nn.Conv2d(n_filters, n_filters*2,
...                             4, 2, 1, bias=False),
...                 nn.BatchNorm2d(n_filters * 2),
...                 nn.LeakyReLU(0.2),
...                 nn.Conv2d(n_filters*2, n_filters*4,
...                             3, 2, 1, bias=False),
...                 nn.BatchNorm2d(n_filters*4),
...                 nn.LeakyReLU(0.2),
...                 nn.Conv2d(n_filters*4, 1, 4, 1, 0, bias=False),
...                 nn.Sigmoid()
...             )
...
...         def forward(self, input):
...             çıktı = self.network(girdi)
...             return output.view(-1, 1).squeeze(0)
```

Yardımcı fonksiyon ve sınıf ile bir DCGAN modeli oluşturabilir ve önceki bölümde basit, tam bağlı GAN'ı uygularken başlattığımız aynı MNIST veri kümesi nesnesini kullanarak eğitebilirsiniz. Yardımcı fonksiyonu kullanarak üretici ağları oluşturabilir ve mimarisini aşağıdaki gibi yazdırabiliriz:

```
>> z_size = 100
>> image_size = (28, 28)
>> n_filters = 32
>>> gen_model = make_generator_network(z_size, n_filters).to(device)
>> print(gen_model) Sıralı(
  (0): ConvTranspose2d(100, 128, kernel_size=(4, 4), stride=(1, 1), bias=False)
  (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
  (2): LeakyReLU(negative_slope=0.2)
  (3): ConvTranspose2d(128, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
  (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
  (5): LeakyReLU(negative_slope=0.2)
  (6): ConvTranspose2d(64, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1), bias=False)
  (7): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
```

 olinpikbarbell@gmail.com

```
  (8) : LeakyReLU(negative_slope=0.2)
  (9) : ConvTranspose2d(32, 1, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1), bias=False)
 (10): Tanh()
)
```

Benzer şekilde, ayırıcı ağı oluşturabilir ve mimarisini görebiliriz:

```
> > >  disk_model = Discriminator(n_filters).to(device)
>> print(disc_model) Ayrıştırıcı(
  (ağ): Sıralı(
    (0): Conv2d(1, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
bias=False)
    (1) : LeakyReLU(negative_slope=0.2)
    (2) : Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
bias=False)
    (3) : BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    (4) : LeakyReLU(negative_slope=0.2)
    (5) : Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
bias=False)
    (6) : BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    (7) : LeakyReLU(negative_slope=0.2)
    (8) : Conv2d(128, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (9) : Sigmoid()
  )
)
```

Ayrıca, *GAN modelinin eğitiminde* yaptığımız gibi aynı kayıp fonksiyonlarını ve optimize edicileri kullanabiliriz
alt bölüm:

```
>> kayıp_fn = nn.BCELoss()
>> g_optimizer = torch.optim.Adam(gen_model.parameters(), 0.0003)
>> d_optimizer = torch.optim.Adam(disc_model.parameters(), 0.0002)
```

Eğitim prosedüründe birkaç küçük değişiklik yapacağız. Rastgele girdi üretmek için kullanılan create_noise() fonksiyonu, vektör yerine dört boyutlu bir tensör çıktısı verecek şekilde

```
>> def create_noise(batch_size, z_size, mode_z):
...     if mode_z == 'uniform':
...         input_z = torch.rand(batch_size, z_size, 1, 1)*2 - 1
...     elif mode_z == 'normal':
...         input_z = torch.randn(batch_size, z_size, 1, 1)
...     return input_z
```

değiştirilmelidir:

Ayırıcıyı eğitmek için kullanılan d_train() işlevinin giriş görüntüsünü yeniden şekillendirmesine gerek yoktur:

```
>> def d_train(x):
...     disc_model.zero_grad()
...     # Ayırıcıyı gerçek bir parti ile eğitin batch_size =
...     x.size(0)
...     x = x.to(cihaz)
...     d_labels_real = torch.ones(batch_size, 1, device=device)
...     d_proba_real = disc_model(x)
...     d_loss_real = loss_fn(d_proba_real, d_labels_real) #
...     Ayırıcıyı sahte bir yığın üzerinde eğitin
...     input_z = create_noise(batch_size, z_size, mode_z).to(device)
...     g_output = gen_model(input_z)
...     d_proba_fake = disc_model(g_output)
...     d_labels_fake = torch.zeros(batch_size, 1, device=device)
...     d_loss_fake = loss_fn(d_proba_fake, d_labels_fake)
...     # gradyan geri yayılımı ve SADECE D'nin parametrelerini
...     optimize edin d_loss = d_loss_real + d_loss_fake
...     d_loss.backward()
...     d_optimizer.step()
...     return d_loss.data.item(), d_proba_real.detach(), \
...             d_proba_fake.detach()
```

olimpikbarbell@gmail.com

Ardından, 100 epok boyunca üretecin ve ayırıcının eğitimi arasında geçiş yapacağız. Her epoktan sonra, create_samples() fonksiyonunu çağırarak mevcut üreteç modelini kullanarak sabit bir gürültü girişinden bazı örnekler üreteceğiz. Kod aşağıdaki gibidir:

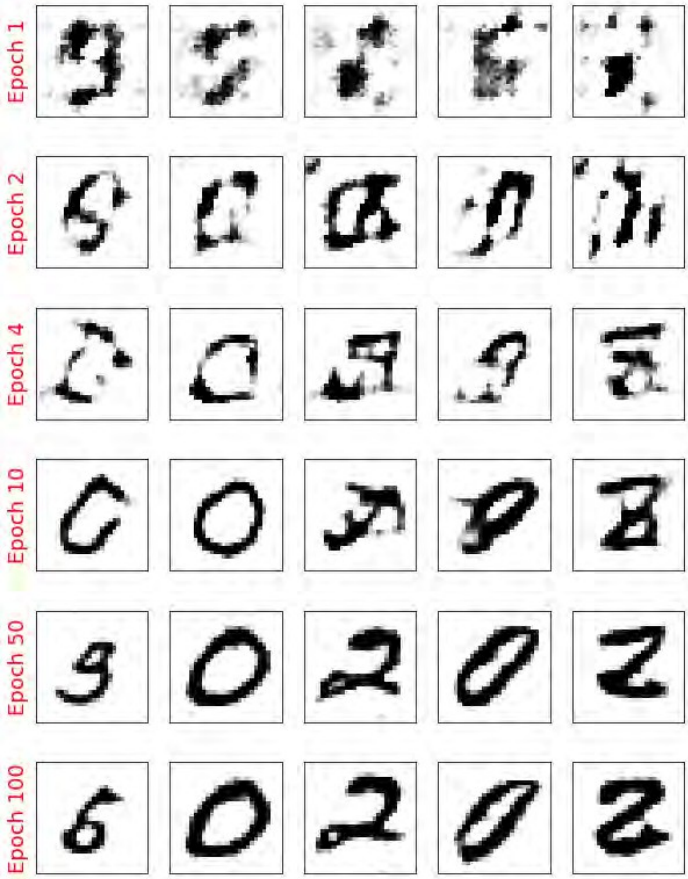
```
>>> fixed_z = create_noise(batch_size, z_size, mode_z).to(device)
>> epoch_samples = []
>>> torch.manual_seed(1)
>> for epoch in range(1, num_epochs+1):
...     gen_model.train()
...     for i, (x, _) in enumerate(mnist_dl):
...         d_loss, d_proba_real, d_proba_fake = d_train(x)
...         d_kayıplar.append(d_kayıp)
...         g_losses.append(g_train(x))
...     print(f'Epoch {epoch:03d} | Ortalama Kayıplar >>'
...           f' G/D {torch.FloatTensor(g_losses).mean():.4f}'
...           f'/{torch.FloatTensor(d_losses).mean():.4f}')
...     gen_model.eval()
...     epoch_samples.append(
...         create_samples(
...             gen_model, sabit_z
...         ).detach().cpu().numpy()
...     )
Dönem 001 | Ortalama Kayıplar >> G/D
4.7016/0.1035 Dönem 002 | Ortalama Kayıplar >>
G/D 5.9341/0.0438
...
Dönem 099 | Ortalama Kayıplar >> G/D
4.3753/0.1360 Dönem 100 | Ortalama Kayıplar >>
G/D 4.4914/0.1120
```

Son olarak, modelin nasıl öğrendiğini ve sentezlenen örneklerin kalitesinin öğrenme süresince nasıl değiştiğini görmek için kaydedilen örnekleri bazı epoklarda görselleştirelim:

```
>> selected_epochs = [1, 2, 4, 10, 50, 100]
>>> fig = plt.figure(figsize=(10, 14))
>> for i,e in enumerate(selected_epochs):
...     for j in range(5):
...         ax = fig.add_subplot(6, 5, i*5+j+1)
...         ax.set_xticks([])
...         ax.set_yticks([])
...         j == 0 ise:
...             ax.text(-0.06, 0.5, f'Epoch {e}',
...                     rotation=90, size=18, color='red',
...                     horizontalalignment='right',
...                     verticalalignment='center',
```

```
...                         transform=ax.transAxes)
...
...             görüntü = epoch_samples[e-1][j]
...             ax.imshow(görüntü, cmap='gray_r')
>> plt.show()
```

Şekil 17.17 sonuçları göstermektedir:



Şekil 17.17: DCGAN'dan üretilen görüntüler

Sonuçları görselleştirmek için vanilya GAN ile ilgili bölümdekiyle aynı kodu kullandık. Yeni örneklerin karşılaştırılması DCGAN'ın çok daha yüksek kalitede görüntüler üretebildiğini göstermektedir.

GAN jeneratörlerinin sonuçlarını nasıl değerlendirebileceğimizi merak edebilirsiniz. En basit yaklaşım, sentezlenen görüntülerin kalitesinin hedef alan ve proje amacı bağlamında değerlendirilmesini içeren görsel değerlendirmedir. Ayrıca, daha az öznel olan ve alan bilgisiyle daha az sınırlı olan birkaç daha sofistike değerlendirme yöntemi önerilmiştir. Ayrıntılı bir inceleme için bkz. *GAN Değerlendirme Ölçütlerinin Artıları ve Eksileri: Yeni Gelişmeler* (https://arxiv.org/abs/2103.09396). Bu makale, jeneratör değerlendirmesini nitel ve nicel ölçütler olarak özetlemektedir.

 ilinpikbarbell@gmail.com

Üretecin eğitiminin, gerçek verilerde gözlemlenen dağılım ile sentezlenen örneklerde gözlemlenen dağılım arasındaki farklılığı en aza indirmeye çalışması gerektiğine dair teorik bir argüman vardır. Dolayısıyla, mevcut mimarimiz çapraz entropiyi bir kayıp fonksiyonu olarak kullandığında çok iyi performans göstermeyecektir.

Bir sonraki alt bölümde, eğitim performansını iyileştirmek için gerçek ve sahte görüntülerin dağılımları arasındaki Wasserstein-1 (veya earth mover's) uzaklığı olarak adlandırılan değiştirilmiş bir kayıp fonksiyonu kullanan WGAN'ı ele alacağız.

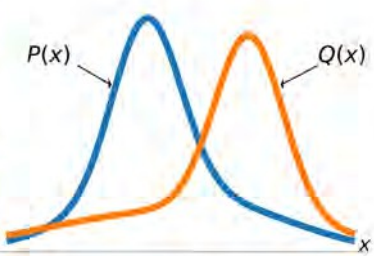
İki dağılım arasındaki farklılık ölçüleri

İlk olarak iki dağılım arasındaki ıraksamayı hesaplamak için farklı ölçütler göreceğiz. Ardından, bu ölçütlerden hangisinin orijinal GAN modelinde zaten gömülü olduğunu göreceğiz. Son olarak, bu ölçütün GAN'larda değiştirilmesi bizi bir WGAN'ın uygulanmasına götürecektir.

Bu bölümün başında belirtildiği gibi, üretken bir modelin amacı, eğitim veri kümesinin dağılımıyla aynı dağılıma sahip yeni örneklerin nasıl sentezleneceğini öğrenmektir. $P(x)$ ve $Q(x)$, aşağıdaki şekilde gösterildiği gibi bir rastgele değişken olan x'in dağılımını temsil etsin.

İlk olarak, *Şekil 17.18'de* gösterilen, P ve Q *gibi* iki dağılım arasındaki farklılığı ölçmek için kullanabileceğimiz bazı yollara bakalım:

Measures	Formulation
Total variation (TV)	$TV(P,Q) = \sup_x \lvert P(x) - Q(x) \rvert$
Kullback-Leibler (KL) divergence	$KL(P\|\|Q) = \int P(x) \log \frac{P(x)}{Q(x)} dx$
Jensen-Shannon (JS) divergence	$JS(P,Q) = \frac{1}{2}\left(KL\left(P\|\|\frac{P+Q}{2}\right) + KL\left(Q\|\|\frac{P+Q}{2}\right)\right)$
Earth mover's (EM) distance	$EM(P,Q) = \inf_{\gamma \in \Pi(P,Q)} E_{(u,v)\in\gamma}(\|u-v\|)$



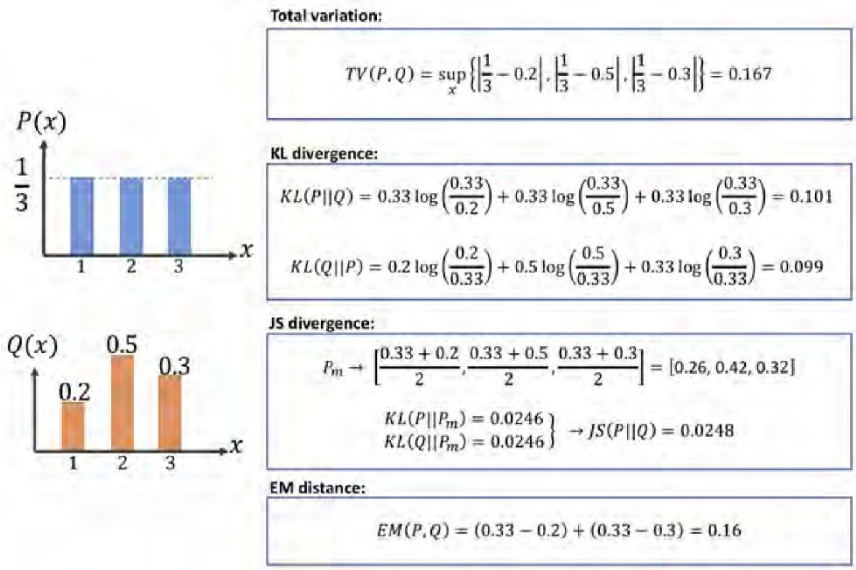
Şekil 17.18: P ve Q dağılımları arasındaki farklılığı ölçme yöntemleri

Toplam varyasyon (TV) ölçüsünde kullanılan supremum, *sup*(*S*) fonksiyonu, *S*'nin tüm elemanlarından büyük olan en küçük değeri ifade eder. Başka bir deyişle, *sup*(S) S için en küçük üst sınırdır. Bunun tersi olarak, EM uzaklığında kullanılan infimum, *inf*(*S*) fonksiyonu, S'nin tüm elemanlarından küçük olan en büyük değeri ifade eder *(en büyük* alt sınır).

Basit kelimelerle neyi başarmaya çalıştıklarını kısaca belirterek bu önlemleri anlayalım:

- Bunlardan ilki olan TV mesafesi, her noktada iki dağılım arasındaki en büyük farkı ölçer.
- EM mesafesi, bir dağılımı diğerine dönüştürmek için gereken minimum iş miktarı olarak yorumlanabilir. EM mesafesindeki infimum fonksiyonu, marjinalleri P veya Q olan tüm ortak dağılımların toplamı olan $\Pi(PP\ \ PP)$ üzerinden alınır. Daha sonra, $\gamma\gamma(uu\ \ v)$, bu tür transferlerden sonra geçerli dağılımları korumak için bazı kısıtlamalara tabi olarak dünyayı u konumundan v konumuna nasıl yeniden dağıttığımızı gösteren bir transfer planıdır. EM mesafesini hesaplamak başlı başına bir optimizasyon problemidir, yani en uygun transfer planını bulmaktır, $\gamma\gamma(uu\ v)$.
- **Kullback-Leibler** (**KL**) ve **Jensen-Shannon** (**JS**) ıraksama ölçüleri bilgi teorisi alanından gelmektedir. KL diverjansının simetrik olmadığını, yani JS diverjansının aksine $KL(P||QQ) \neq KL(QQ||P)$ olduğunu unutmayın.

*Şekil 17.18'*de verilen benzerlik denklemleri sürekli dağılımlara karşılık gelmektedir ancak ayrık durumlar için genişletilebilir. Bu farklı benzerlik ölçülerinin iki basit ayrık dağılımla hesaplanmasına ilişkin bir örnek *Şekil 17.19'da* gösterilmektedir:



Şekil 17.19: Farklı benzerlik ölçülerinin hesaplanmasına bir örnek

EM mesafesi durumunda, bu basit örnek için, $x = 2$'de $Q(x)$'in şu değerlere sahip olduğunu görebiliriz fazla değeri $0.5 - \frac{1}{3} = 0{,}166$ iken, diğer iki x'teki Q değeri 1/3'ün altındadır. Bu nedenle, Q
En az iş miktarı, *Şekil'*de gösterildiği gibi $x = 2$'deki ekstra değeri $x = 1$ ve $x = 3$'e aktardığımızda ortaya çıkar.
17.19. Bu basit örnek için, bu aktarımların tüm olası aktarımlar arasında en az miktarda işle sonuçlanacağını görmek kolaydır. Ancak, daha karmaşık durumlar için bunu yapmak mümkün olmayabilir.

 ilinpikbarbell@gmail.com

KL ıraksaması ve çapraz entropi arasındaki ilişki

KL diverjansı, $KL(P||QQ)$, P dağılımının Q referans dağılımına g ö r e göreli entropisini ölçer. KL diverjansı için formülasyon şu şekilde genişletilebilir:

$$KL(P||QQ) = -\int P(x) \log(QQ(x))\, dx - (-\int P(x) \log(P(x)))$$

Ayrıca, ayrık dağılımlar için KL ıraksaması şu şekilde yazılabilir:

$$KL(P||QQ) = -\sum_i P(x_i) \log\left(\frac{P(x_i)}{QQ(x_i)}\right)$$

olarak benzer şekilde genişletilebilir:

$$KL(P||QQ) = -\sum_i P(x_i) \log(QQ(x_i)) - (-\sum_i P(x_i) \log(P(x_i)))$$

Genişletilmiş formülasyona (ayrık veya sürekli) dayalı olarak KL ıraksaması, P ve Q arasındaki çapraz entropi (önceki denklemdeki ilk terim) i l e *P'nin (*öz) entropisinin (ikinci terim) çıkarılması olarak görülür, Yani, $KL(P||QQ) = H(PP\ QQ) - H(P)$.

Şimdi, GAN'larla ilgili tartışmamıza geri dönerek, bu farklı mesafe ölçülerinin GAN'lar için kayıp fonksiyonuyla nasıl ilişkili olduğunu görelim. Orijinal GAN'daki kayıp fonksiyonunun gerçekten de *gerçek ve sahte örneklerin dağılımı arasındaki JS farklılığını en aza indirdiği* matematiksel *olarak* gösterilebilir. Ancak, *Martin Arjovsky* ve meslektaşlarının bir makalesinde tartışıldığı gibi (*Wasserstein Generative Adversarial Networks*, http:// proceedings.mlr.press/v70/arjovsky17a/arjovsky17a.pdf), JS sapmasının bir GAN modelini eğitmede sorunları vardır ve bu nedenle, eğitimi iyileştirmek için araştırmacılar, gerçek ve sahte örneklerin dağılımı arasındaki farklılığın bir ölçüsü olarak EM mesafesini kullanmayı önermişlerdir.

EM mesafesini kullanmanın avantajı nedir?

Bu soruyu yanıtlamak için Martin Arjovsky ve arkadaşlarının daha önce bahsedilen makalesinde verilen bir örneği ele alabiliriz. Basitçe ifade etmek gerekirse, iki paralel doğru olan P ve Q *olmak* üzere iki dağılımımız olduğunu varsayalım. Doğrulardan biri $x = 0$'da sabittir ve diğer doğru x ekseni boyunca hareket edebilir ancak başlangıçta x x xx'de bulunur, burada $\theta\theta$ $\Theta\Theta$.

KL, TV ve JS benzerlik ölçülerinin $KL(P||QQ) = +\infty$ olduğu gösterilebilir, parametresine göre farklılaştırılamaz ve bu nedenle P ve Q dağılımlarını birbirine benzer hale getirecek şekilde $\Theta\Theta$'ya göre farklılaştırılamazlar. Öte yandan, EM mesafesi $EM(PP\ PP) = |\theta\theta|$ olup, $\theta\theta$'ya göre gradyanı mevcuttur ve Q'yu P'ye doğru itebilir.

$TV(PP\ PP) = 1$ ve $JS(PP\ PP) = \log 2$. Bu benzerlik ölçülerinin hiçbiri bir fonksiyon değildir

Şimdi dikkatimizi EM mesafesinin bir GAN modelini eğitmek için nasıl kullanılabileceğine odaklayalım. P_r 'nin gerçek örneklerin dağılımı olduğunu ve P_g 'nin sahte (üretilmiş) örneklerin dağılımını gösterdiğini varsayalım. P_r ve P_g EM mesafesi denkleminde P ve Q'nun yerini alır. Daha önce de belirtildiği gibi, EM mesafesini hesaplamak başlı başına bir optimizasyon problemidir; bu nedenle, özellikle de bu hesaplamayı GAN eğitim döngüsünün her bir iterasyonunda tekrarlamak istiyorsak, hesaplama açısından zorlayıcı hale gelir. Neyse ki EM mesafesinin hesaplanması **Kantorovich-Ru- binstein dualitesi** adı verilen bir teorem kullanılarak aşağıdaki gibi basitleştirilebilir:

$$W(P_r, P_g) = \sup_{\|f\|_L \le 1} \mathbb{E}_{u \sim P_r}[f(u)] - \mathbb{E}_{v \sim P_g}[f(v)]$$

Burada supremum, $\|f\|_L \le 1$ ile gösterilen tüm 1-Lipschitz sürekli fonksiyonlar üzerinden alınır.

Lipschitz sürekliliği

1-Lipschitz sürekliliğine dayanarak, f fonksiyonu aşağıdaki özelliği sağlamalıdır:

$$|f(x_1) - f(x_2)| \le |x_1 - x|_2$$

Ayrıca, $f: R \rightarrow R$ özelliğini sağlayan gerçel bir fonksiyon

$$|f(x_1) - f(x_2)| \le K|x_1 - x|_2$$

K-Lipschitz sürekliliği olarak adlandırılır.

GAN'lar için pratikte EM mesafesini kullanma

Şimdi soru şu: Bir GAN için gerçek (P_r) ve sahte (P_g) çıktıların dağılımı arasındaki Wasserstein mesafesini hesaplamak için böyle bir 1-Lipschitz sürekli fonksiyonu nasıl buluruz? WGAN yaklaşımının arkasındaki teorik kavramlar ilk başta karmaşık görünse de, bu sorunun cevabı göründüğünden daha basittir. Derin NN'lerin evrensel fonksiyon yaklaşımcıları olduğunu düşündüğümüzü hatırlayın. Bu, bir NN modelini Wasserstein mesafe fonksiyonuna yaklaştırmak için basitçe eğitebileceğimiz anlamına gelir. Önceki bölümde gördüğünüz gibi, basit GAN sınıflandırıcı biçiminde bir ayırıcı kullanır. WGAN için ayırıcı, bir olasılık değeri yerine skaler bir puan döndüren bir *eleştirmen* gibi davranacak şekilde değiştirilebilir. Bu puanı, girdi görüntülerinin ne kadar gerçekçi olduğu şeklinde yorumlayabiliriz (bir sanat eleştirmeninin galerideki sanat eserlerine puan vermesi gibi).

Wasserstein mesafesini kullanarak bir GAN'ı eğitmek için, ayırıcı, D, ve üreteç, G, için kayıplar aşağıdaki gibi tanımlanır. Eleştirici (yani ayırıcı ağ), gerçek görüntü örnekleri yığını ve sentezlenmiş örnekler yığını için çıktılarını döndürür. Sırasıyla $D(\boldsymbol{x})$ ve $D(G(\boldsymbol{z}))$ notasyonlarını kullanıyoruz.

 olimpikbarbell@gmail.com

Ardından, aşağıdaki kayıp terimleri tanımlanabilir:

- Ayırıcının kaybının gerçek bileşeni:

$$L^{D}_{\text{gerçek}} = \frac{1}{N}\sum_{i} D(x_i)$$

- Ayırıcının kaybının sahte bileşeni:

$$L^{D}_{\text{sahte}} = \frac{1}{N}\sum_{i} D(G(z_i))$$

- Jeneratör için kayıp:

$$L^{G} = -\frac{1}{N}\sum_{i} D(G(z_i))$$

WGAN için hepsi bu kadar, ancak eğitim sırasında kritik fonksiyonun 1-Lipschitz özelliğinin korunmasını sağlamamız gerekiyor. Bu amaçla, WGAN makalesi ağırlıkların küçük bir bölgeye, örneğin [-0.01, 0.01]'e sıkıştırılmasını önermektedir.

Gradyan cezası

Arjovsky ve arkadaşlarının makalesinde, ayırıcının (veya eleştiricinin) 1-Lipschitz özelliği için ağırlık kırpma önerilmektedir. Ancak, *Ishaan Gulrajani* ve meslektaşları tarafından hazırlanan ve https://arxiv.org/pdf/1704.00028. pdf adresinden ücretsiz olarak erişilebilen *Wasserstein GAN*'larının *Geliştirilmiş Eğitimi (*2017) başlıklı başka bir makalede, Ishaan Gulrajani ve meslektaşları, ağırlıkların kırpılmasının patlayan ve kaybolan gradyanlara yol açabileceğini göstermiştir. Ayrıca, ağırlıkların kırpılması kapasitenin yetersiz kullanımına da yol açabilir; bu da eleştirmen ağının daha karmaşık işlevlerin aksine yalnızca bazı basit işlevleri öğrenmekle sınırlı kalması anlamına gelir. Bu nedenle, Ishaan Gulrajani ve meslektaşları ağırlıkları **kırpmak** yerine alternatif bir çözüm olarak gradyan **cezasını** (**GP)** önermişlerdir. Sonuç olarak **gradyan cezalı WGAN** (**WGAN-GP**) ortaya çıkmıştır.

Her iterasyonda eklenen GP için prosedür aşağıdaki adımlar dizisi ile özetlenebilir:

1. Belirli bir partideki her bir gerçek ve sahte örnek çifti ($x^{[i]}, \tilde{x}^{[i]}$) için rastgele bir sayı seçin, $\alpha^{[i]}$, tekdüze bir dağılımdan örneklenir, yani $\alpha^{[i]} \in U(0,1)$.
2. Gerçek ve sahte örnekler arasında bir enterpolasyon hesaplayın: $\check{x}^{[i]} = \alpha x^{[i]} + (1-\alpha)\tilde{x}^{[i]}$, yeniden enterpole edilmiş örneklerin bir yığını ile sonuçlanır.
3. Tüm enterpole edilmiş örnekler için ayırıcı (eleştirmen) çıktısını hesaplayın, $D(D(\check{x}^{[i]}))$.
4. Eleştiricinin çıktısının her bir enterpolasyonlu örneğe göre gradyanlarını hesaplayın, yani $\nabla_{\check{x}^{[i]}} D(D(\check{x}^{[i]}))$.
5. GP'yi şu şekilde hesaplayın:

$$L^{D}_{gp} = \frac{1}{N}\sum_{i} \left(\left\|\nabla_{\check{x}^{[i]}} D(\check{x}^{[i]})\right\|_2 - 1\right)^2$$

Ayırıcı için toplam kayıp aşağıdaki gibidir:

$$L^D_{\text{toplam}} = L^D_{\text{gerçek}} + L^D_{\text{sahte}} + \lambda L^D_{gp}$$

Burada, λ ayarlanabilir bir hiperparametredir.

DCGAN modelini eğitmek için WGAN-GP'nin uygulanması

DCGAN için üretici ve ayırıcı ağları oluşturan yardımcı fonksiyonu ve sınıfı zaten tanımlamıştık (make_generator_network() ve Discriminator()). WGAN'da toplu normalleştirme yerine katman normalleştirmenin kullanılması önerilir. Katman normalizasyonu, toplu normalizasyondaki toplu boyut yerine girdileri özellikler arasında normalleştirir. WGAN modelini oluşturmak için kullanılan kod aşağıdaki gibidir:

```
>>> def make_generator_network_wgan(input_size, n_filters):
...     model = nn.Sequential(
...         nn.ConvTranspose2d(input_size, n_filters*4, 4,
...                            1, 0, bias=False),
...         nn.InstanceNorm2d(n_filters*4),
...         nn.LeakyReLU(0.2),
...
...         nn.ConvTranspose2d(n_filters*4, n_filters*2,
...                            3, 2, 1, bias=False),
...         nn.InstanceNorm2d(n_filters*2),
...         nn.LeakyReLU(0.2),
...
...         nn.ConvTranspose2d(n_filters*2, n_filters, 4,
...                            2, 1, bias=False),
...         nn.InstanceNorm2d(n_filters),
...         nn.LeakyReLU(0.2),
...
...         nn.ConvTranspose2d(n_filters, 1, 4, 2, 1, bias=False),
...         nn.Tanh()
...     )
...     dönüş modeli
>>>
>>> class DiscriminatorWGAN(nn.Module):
...     def init (self, n_filters):
...         super(). init ()
...         self.network = nn.Sequential(
...             nn.Conv2d(1, n_filters, 4, 2, 1, bias=False),
...             nn.LeakyReLU(0.2),
...
```

 olimpikbarbell@gmail.com

```
...             nn.Conv2d(n_filters, n_filters*2, 4, 2, 1,
...                       bias=False),
...             nn.InstanceNorm2d(n_filters * 2),
...             nn.LeakyReLU(0.2),
...
...             nn.Conv2d(n_filters*2, n_filters*4, 3, 2, 1,
...                       bias=False),
...             nn.InstanceNorm2d(n_filters*4),
...             nn.LeakyReLU(0.2),
...
...             nn.Conv2d(n_filters*4, 1, 4, 1, 0, bias=False),
...             nn.Sigmoid()
...         )
...
...         def forward(self, input):
...             çıktı = self.network(girdi)
...             return output.view(-1, 1).squeeze(0)
```

Şimdi ağları ve optimize edicilerini aşağıdaki gibi başlatabiliriz:

```
>>> gen_model = make_generator_network_wgan(
...         z_size, n_filters
... ).to(cihaz)
> > >   disk_model = DiscriminatorWGAN(n_filters).to(device)
>> g_optimizer = torch.optim.Adam(gen_model.parameters(), 0.0002)
>> d_optimizer = torch.optim.Adam(disc_model.parameters(), 0.0002)
```

Daha sonra, GP bileşenini hesaplayacak fonksiyonu aşağıdaki gibi tanımlayacağız:

```
>> from torch.autograd import grad as torch_grad
>> def gradient_penalty(real_data, generated_data):
...         batch_size = real_data.size(0)
...
...         # Enterpolasyonu hesapla
...         alpha = torch.rand(real_data.shape[0], 1, 1, 1,
...                            requires_grad=True,
...         device=device) interpolated = alpha * real_data + \
...                        (1 - alfa) * üretilen_veri
...
...         # Enterpole edilmiş örneklerin olasılığını hesaplayın
...         proba_interpolated = disc_model(interpolated)
...
```

```
...     # Olasılıkların gradyanlarını hesaplayın
...     gradients = torch_grad(
...         outputs=proba_interpolated, inputs=interpolated,
...         grad_outputs=torch.ones(proba_interpolated.size(),
...                                 device=device),
...         create_graph=True, retain_graph=True
...     )[0]
...
...     gradients = gradients.view(batch_size, -1)
...     gradients_norm = gradients.norm(2, dim=1)
...     lambda_gp * ((gradients_norm - 1)**2).mean()
...     döndürür
```

Ayırıcı ve üreteç eğitim fonksiyonlarının WGAN versiyonu aşağıdaki gibidir:

```
>> def d_train_wgan(x):
...     disc_model.zero_grad()
...
...     batch_size = x.size(0)
...     x = x.to(cihaz)
...
...     # Gerçek ve oluşturulmuş veriler üzerinde olasılıkları
...     hesaplayın d_real = disc_model(x)
...     input_z = create_noise(batch_size, z_size, mode_z).to(device)
...     g_output = gen_model(input_z)
...     d_oluşturulan = disk_modeli(g_çıktı)
...     d_loss = d_generated.mean() – d_real.mean() + \
...                 gradient_penalty(x.data, g_output.data)
...     d_loss.backward()
...     d_optimizer.step()
...     return d_loss.data.item()
>>>
>> def g_train_wgan(x):
...     gen_model.zero_grad()
...
...     batch_size = x.size(0)
...     input_z = create_noise(batch_size, z_size, mode_z).to(device)
...     g_output = gen_model(input_z)
...
...     d_üretilen = disk_modeli(g_çıktı) g_kayıp = -
...     d_üretilen.ortalama()
...
```

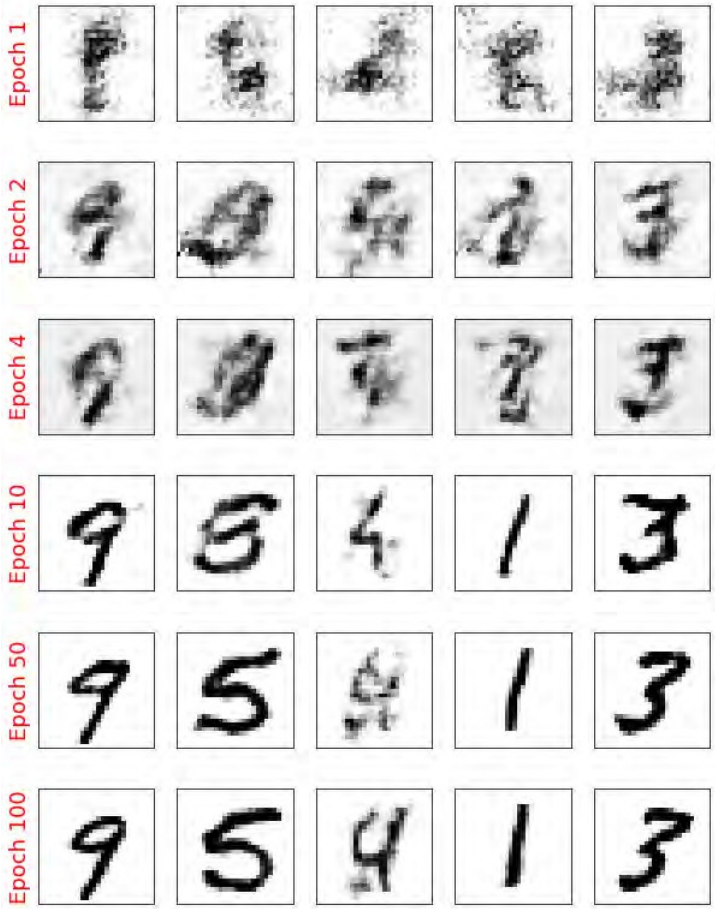
 olimpikbarbell@gmail.com

```
...     # gradient backprop ve SADECE G'nin parametrelerini
        optimize edin
...     g_loss.backward()
...     g_optimizer.step() return
...     g_loss.data.item()
```

Daha sonra modeli 100 epok için eğiteceğiz ve sabit bir gürültü girişinin jeneratör çıkışını kaydedeceğiz:

```
>>> epoch_samples_wgan = []
>> lambda_gp = 10.0
>> num_epochs = 100
>>> torch.manual_seed(1)
>>  critic_iterations = 5
>> for epoch in range(1, num_epochs+1):
...     gen_model.train()
...     d_losses, g_losses = [], []
...     for i, (x, _) in enumerate(mnist_dl):
...         for _ in range(critic_iterations):
...             d_loss = d_train_wgan(x)
...         d_losses.append(d_loss)
...         g_losses.append(g_train_wgan(x))
...
...     print(f'Epoch {epoch:03d} | D Kaybı >>'
...           f' {torch.FloatTensor(d_losses).mean():.4f}')
...     gen_model.eval()
...     epoch_samples_wgan.append(
...         create_samples(
...             gen_model, sabit_z
...         ).detach().cpu().numpy()
...     )
```

Son olarak, WGAN modelinin nasıl öğrendiğini ve sentezlenen örneklerin kalitesinin öğrenme süresince nasıl değiştiğini görmek için kaydedilen örnekleri bazı epoklarda görselleştirelim. Aşağıdaki şekil, DCGAN modelinin ürettiğinden biraz daha iyi görüntü kalitesi gösteren sonuçları göstermektedir:



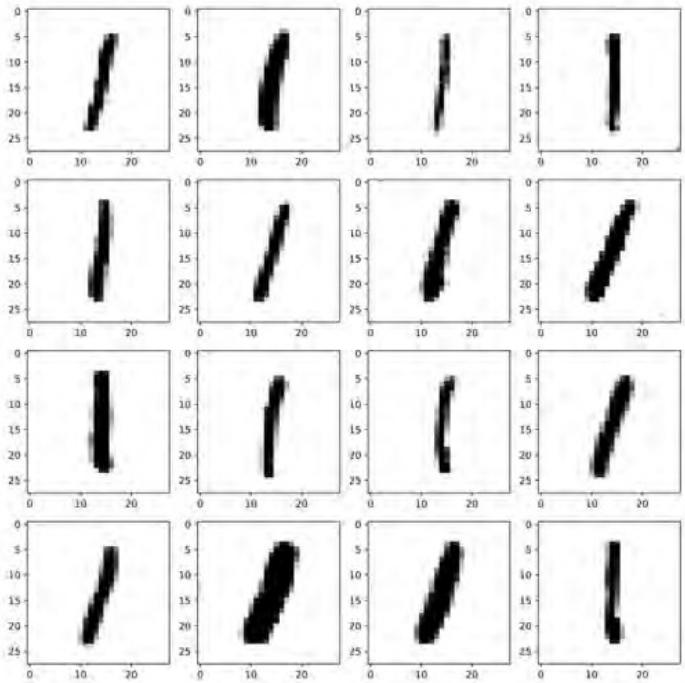
Şekil 17.20: WGAN kullanılarak oluşturulan görüntüler

Mod çökmesi

GAN modellerinin düşmanca doğası nedeniyle, onları eğitmek oldukça zordur. GAN'ların eğitiminde başarısızlığın yaygın bir nedeni, üretecin küçük bir alt uzayda sıkışıp kalması ve benzer örnekler üretmeyi öğrenmesidir. Buna **mod çökmesi** denir ve bir örneği *Şekil 17.21'de* gösterilmiştir.

Bu şekildeki sentezlenmiş örnekler seçilmemiştir. Bu, üretecin tüm veri dağılımını öğrenemediğini ve bunun yerine bir alt uzaya odaklanan tembel bir yaklaşım benimsediğini göstermektedir:

 olinpikbarbell@gmail.com



Şekil 17.21: Mod çökmesine bir örnek

Daha önce gördüğümüz kaybolan ve patlayan gradyan sorunlarının yanı sıra, GAN modellerinin eğitimini zorlaştırabilecek bazı başka hususlar da vardır (aslında bu bir sanattır). İşte GAN sanatçılarının önerdiği birkaç püf noktası.

Yaklaşımlardan biri, yalnızca gerçek veya sahte örneklerden oluşan yığınların ayrımcıya ayrı ayrı beslenmesine dayanan **mini yığın ayrımcılığı** olarak adlandırılır. Mini parti ayrımcılığında, bir partinin gerçek mi yoksa sahte mi olduğunu görmek için ayrımcının bu partilerdeki örnekleri karşılaştırmasına izin veririz. Bir modelin mod çöküşünden muzdarip olması durumunda, yalnızca gerçek örneklerden oluşan bir yığının çeşitliliği büyük olasılıkla sahte bir yığının çeşitliliğinden daha yüksektir.

GAN eğitimini stabilize etmek için yaygın olarak kullanılan bir diğer teknik de *özellik eşleştirmedir*. Özellik eşleştirmede, ayrıştırıcının ara temsillerine (özellik haritaları) dayalı olarak orijinal ve sentezlenmiş görüntüler arasındaki farkı en aza indiren fazladan bir terim ekleyerek üretecin amaç fonksiyonunda küçük bir değişiklik yaparız. Bu teknik hakkında daha fazla bilgiyi *Ting-Chun Wang* ve meslektaşlarının https://arxiv. org/pdf/1711.11585.pdf adresinden ücretsiz olarak erişilebilen *High Resolution Image Syn- thesis and Semantic Manipulation with Conditional GANs* başlıklı orijinal makalesinde okumanızı öneririz.

Eğitim sırasında, bir GAN modeli de birkaç modda takılıp kalabilir ve bunlar arasında atlayabilir. Bu davranışı önlemek için, bazı eski örnekleri saklayabilir ve üretecin önceki modları tekrar ziyaret etmesini önlemek için bunları ayırıcıya besleyebilirsiniz. Bu teknik *deneyim tekrarı* olarak adlandırılır. Ayrıca, birden fazla GAN'ı farklı rastgele tohumlarla eğitebilirsiniz, böylece hepsinin kombinasyonu veri dağılımının herhangi birinden daha büyük bir bölümünü kapsar.

Diğer GAN uygulamaları

Bu bölümde, temel olarak GAN'ları kullanarak örnekler üretmeye odaklandık ve sentezlenen çıktıların kalitesini artırmak için birkaç püf noktası ve tekniği inceledik. GAN'ların uygulamaları bilgisayarla görme, makine öğrenimi ve hatta diğer bilim ve mühendislik alanları da dahil olmak üzere hızla genişlemektedir. Farklı GAN modellerinin ve uygulama alanlarının güzel bir listesi https://github. com/hindupuravinash/the-gan-zoo adresinde bulunabilir.

GAN'ları denetimsiz bir şekilde ele aldığımızı belirtmek gerekir; yani, bu bölümde ele alınan modellerde sınıf etiketi oluşumu kullanılmamıştır. Ancak, GAN yaklaşımı yarı denetimli ve denetimli görevler için de genelleştirilebilir. Örneğin, *Mehdi Mirza* ve *Simon Osindero* tarafından *Conditional Generative Adversarial Nets,* 2014 (https://arxiv.org/pdf/1411.1784.pdf) adlı makalede önerilen koşullu **GAN** (**cGAN),** sınıf etiketi bilgisini kullanır ve sağlanan etikete, yani $\boldsymbol{x}\tilde{\boldsymbol{x}} = G(z|y)$ koşuluna bağlı olarak yeni görüntüler sentezlemeyi öğrenir - MNIST'e uygulanır. Bu, seçici olarak 0-9 aralığında farklı rakamlar üretmemizi sağlar. Ayrıca, koşullu GAN'lar görüntüden görüntüye çeviri yapmamıza, yani belirli bir görüntüyü belirli bir alandan diğerine nasıl dönüştüreceğimizi öğrenmemize olanak tanır. Bu bağlamda, ilginç bir çalışma, *Philip Isola* ve meslektaşlarının 2018 tarihli *Image-to-Image Translation with Conditional Adversarial Networks* adlı makalesinde yayınlanan Pix2Pix algoritmasıdır (https://arxiv.org/pdf/1611.07004.pdf). Pix2Pix algoritmasında, ayırt edicinin tüm görüntü için tek bir tahmin yerine görüntüdeki birden fazla yama için gerçek/sahte tahminleri sağladığını belirtmek gerekir.

CycleGAN, cGAN'ın üzerine inşa edilen ve görüntüden görüntüye aktarım için de kullanılan bir başka ilginç GAN modelidir. Ancak, CycleGAN'da iki alandan gelen eğitim örneklerinin eşleştirilmediğini, yani girdiler ve çıktılar arasında bire bir örtüşme olmadığını unutmayın. Örneğin, bir CycleGAN kullanarak yazın çekilen bir resmin mevsimini kış olarak değiştirebiliriz. *Jun-Yan Zhu* ve arkadaşlarının 2020 tarihli *Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks* makalesinde (https://arxiv.org/pdf/1703.10593.pdf) etkileyici bir örnek olarak atların zebralara dönüştürülmesi gösterilmektedir.

Özet

Bu bölümde ilk olarak derin öğrenmedeki üretici modeller ve bunların genel amacı olan yeni verilerin sentezlenmesi hakkında bilgi edindiniz. Daha sonra GAN modellerinin, birbirlerini geliştirmek için düşmanca bir eğitim ortamında birbirleriyle rekabet eden bir üretici ağı ve bir ayırt edici ağı nasıl kullandığını ele aldık. Daha sonra, hem üretici hem de ayırıcı için yalnızca tam bağlı katmanlar kullanan basit bir GAN modeli uyguladık.

GAN modellerinin nasıl geliştirilebileceğini de ele aldık. İlk olarak, hem üretici hem de ayırıcı için derin konvülsiyonel ağlar kullanan bir DCGAN gördünüz. Bu arada, iki yeni kavram hakkında da bilgi edindiniz: transpoze konvolüsyon (özellik haritalarının uzamsal boyutluluğunu örneklemek için) ve BatchNorm (eğitim sırasında yakınsamayı iyileştirmek için).

Daha sonra gerçek ve sahte örneklerin dağılımları arasındaki mesafeyi ölçmek için EM mesafesini kullanan bir WGAN'a baktık. Son olarak, ağırlıkları kırpmak yerine 1-Lipschitz özelliğini korumak için GP ile WGAN hakkında konuştuk.

 olimpikbarbell@gmail.com

Bir sonraki bölümde, çizge sinir ağlarına bakacağız. Daha önce, tablo ve görüntü veri kümelerine odaklanmıştık. Buna karşılık, çizge sinir ağları, sosyal bilimler, mühendislik ve biyolojide her yerde bulunan veri kümeleriyle çalışmamıza olanak tanıyan çizge yapılı veriler için tasarlanmıştır. Çizge yapılı verilerin popüler örnekleri arasında sosyal ağ grafikleri ve kovalent bağlarla birbirine bağlanmış atomlardan oluşan moleküller yer almaktadır.

Kitabımızın Discord alanına katılın

Benzer düşünen insanlarla tanışmak ve 2000'den fazla üyemizle birlikte öğrenmek için Discord topluluğumuza katılın:
https://packt.link/MLwPyTorch

6587191CA584E
olimpikbarbell@gmail.com

18

Çizge Yapılı Verilerde Bağımlılıkları Yakalamak için Çizge Sinir Ağları

Bu bölümde, çizge verileri üzerinde çalışan bir derin öğrenme modeli sınıfını, yani **çizge sinir ağlarını** (**GNN'ler**) tanıtacağız. GNN'ler son yıllarda hızlı bir gelişim alanı olmuştur. Yapay Zekanın *Durumu* 2021 raporuna göre (https://www.stateof.ai/2021-report-launch.html), GNN'ler "nişten yapay zeka araştırmalarının en sıcak alanlarından birine" dönüşmüştür.

GNN'ler aşağıdakiler de dahil olmak üzere çeşitli alanlarda uygulanmıştır:

- Metin sınıflandırma (https://arxiv.org/abs/1710.10903)
- Tavsiye sistemleri (https://arxiv.org/abs/1704.06803)
- Trafik tahmini (https://arxiv.org/abs/1707.01926)
- İlaç keşfi (h ttps://arxiv.org/abs/1806.02473)

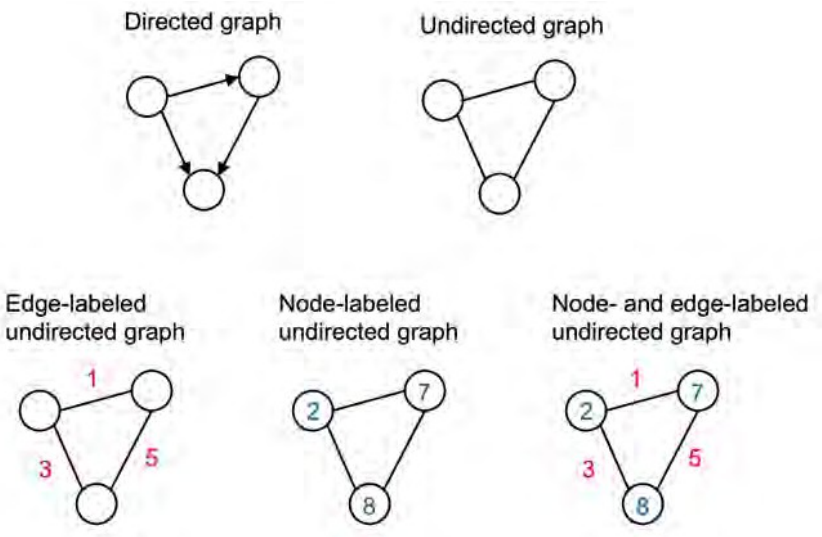
Hızla gelişen bu alandaki her yeni fikri ele alamasak da, GNN'lerin nasıl çalıştığını ve nasıl uygulanabileceğini anlamak için bir temel sağlayacağız. Ayrıca, derin öğrenme için grafik verilerini yönetmek için kaynakların yanı sıra derin öğrenme modellerinizde kullanabileceğiniz birçok farklı grafik katmanının uygulamalarını sağlayan **PyTorch Geometric** kütüphanesini tanıtacağız.

Bu bölümde ele alınacak konular aşağıdaki gibidir:

- Grafik verilerine ve bunların derin sinir ağlarında kullanılmak üzere nasıl temsil edilebileceğine giriş
- Yaygın GNN'lerin önemli bir yapı taşı olan grafik konvolüsyonlarının açıklaması
- PyTorch Geometric kullanarak moleküler özellik tahmini için GNN'lerin nasıl uygulanacağını gösteren bir eğitim
- GNN alanının en ileri noktasındaki yöntemlere genel bir bakış

Grafik verilerine giriş

Genel olarak grafikler, verilerdeki ilişkileri tanımladığımız ve yakaladığımız belirli bir yolu temsil eder. Graflar, doğrusal olmayan ve soyut olan özel bir veri yapısı türüdür. Graflar soyut nesneler olduğundan, graflar üzerinde işlem yapılabilmesi için somut bir temsilin tanımlanması gerekir. Ayrıca, grafikler farklı gösterimler gerektirebilecek belirli özelliklere sahip olacak şekilde tanımlanabilir. *Şekil 18.1,* aşağıdaki alt bölümlerde daha ayrıntılı olarak tartışacağımız yaygın grafik türlerini özetlemektedir:

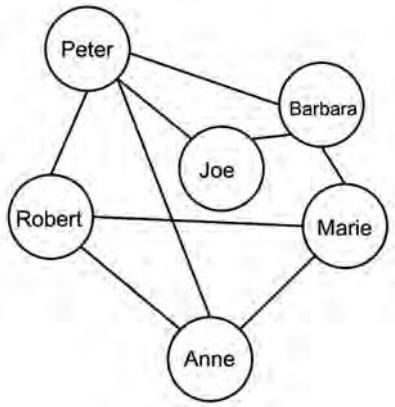


Şekil 18.1: Yaygın grafik türleri

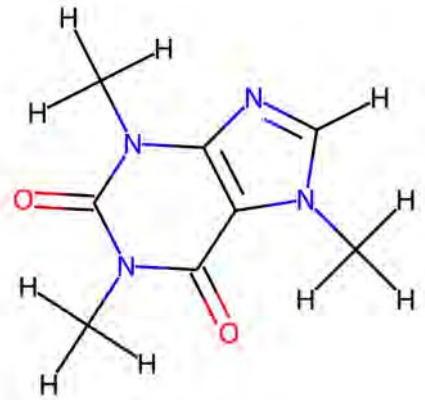
Yönlendirilmemiş grafikler

Yönlendirilmemiş bir **çizge,** düğümlerin sırasının ve bağlantılarının önemli olmadığı kenarlar aracılığıyla birbirine bağlanan düğümlerden (çizge teorisinde genellikle **köşe olarak** da adlandırılır) oluşur. *Şekil 18.2,* yönlendirilmemiş grafların iki tipik örneğini, bir arkadaş grafını ve kimyasal bağlarla bağlanmış atomlardan oluşan bir kimyasal molekül grafını çizmektedir (bu tür moleküler grafları daha sonraki bölümlerde çok daha ayrıntılı olarak tartışacağız):

olimpikbarbell@gmail.com



Friend graph



Molecular graph of caffeine

Şekil 18.2: İki yönlendirilmemiş çizge örneği

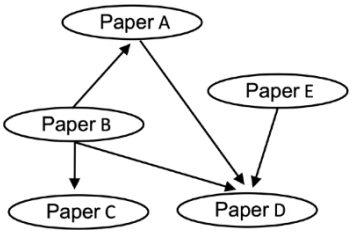
Yönlendirilmemiş grafikler olarak temsil edilebilen diğer yaygın veri örnekleri arasında görüntüler, pro- tein-protein etkileşim ağları ve nokta bulutları yer alır.

Matematiksel olarak, yönlendirilmemiş bir G grafı bir (V, E) çiftidir; burada V grafın düğümlerinin bir kümesidir ve E eşleştirilmiş düğümleri oluşturan kenarların kümesidir. Çizge daha sonra bir |V|×|V| **bitişiklik matrisi** $\boldsymbol{A}$ olarak kodlanabilir. $\boldsymbol{A}$ matrisindeki her x_{ij} elemanı ya 1 ya da 0'dır, 1 i ve j düğümleri arasında bir kenar olduğunu gösterir (tersi, 0 bir kenarın olmadığını gösterir). Çizge yönlendirilmemiş olduğundan, ***A'nın*** ek bir özelliği de $x_{ij} = x$ *olmasıdır*.$_{ji}$

Yönlendirilmiş grafikler

Yönlendirilmiş **çizgeler,** önceki bölümde ele alınan *yönlendirilmemiş çizgelerin* aksine, düğümleri *yönlendirilmiş* kenarlar aracılığıyla birbirine bağlar. Matematiksel olarak, kenar kümesi olan *E'nin sıralı* çiftler kümesi olması dışında, yönlendirilmemiş bir çizge ile aynı şekilde tanımlanırlar. Bu nedenle, ***A'***nın x_{ij} öğesinin x_{ji} öğesine eşit olması gerekmez.

Yönlendirilmiş grafiğin bir örneği, düğümlerin yayınlar olduğu ve bir düğümden gelen kenarların belirli bir makalenin alıntı yaptığı makalelerin düğümlerine yönlendirildiği bir atıf ağıdır.



Şekil 18.3: Bir yönlendirilmiş çizge örneği

Etiketli grafikler

Üzerinde çalışmak istediğimiz birçok grafik, düğüm ve kenarlarının her biriyle ilişkili ek bilgilere sahiptir. Örneğin, daha önce gösterilen kafein molekülünü ele alırsak, moleküller her düğümün bir kimyasal element (örneğin, O, C, N veya H atomları) ve her kenarın iki düğüm arasındaki bağ türü (örneğin, tek veya çift bağ) olduğu grafikler olarak temsil edilebilir. Bu düğüm ve kenar özelliklerinin belli bir kapasitede kodlanması gerekir. Düğüm kümesi ve kenar kümesi ikilisi (V, E) ile tanımlanan G grafiği göz önüne alındığında, bir $|V| \times f_V$ düğüm özellik matrisi $\boldsymbol{X}$ tanımlarız; burada f_V her düğümün etiket vektörünün uzunluğudur. Kenar etiketleri için, bir $|E| \times f_E$ kenar özellik matrisi $\boldsymbol{X}_E$ tanımlarız; burada f_E her bir kenarın etiket vektörünün uzunluğudur.

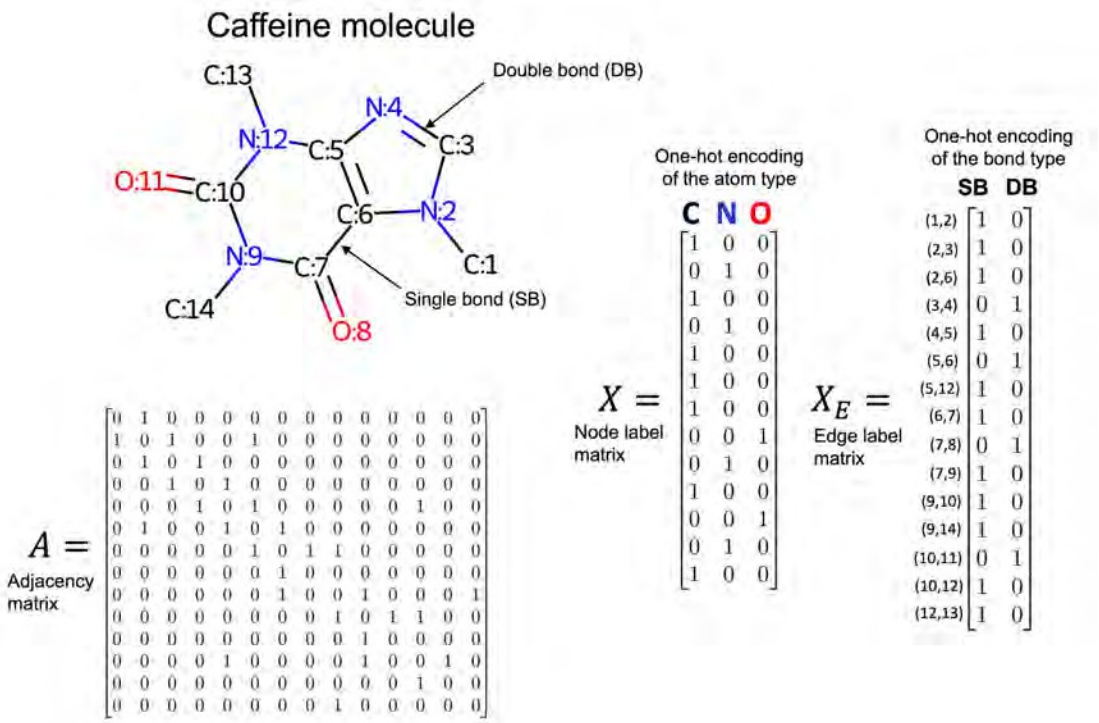
Moleküller, **etiketli** bir **grafik** olarak temsil edilebilen verilere mükemmel bir örnektir ve bölüm boyunca moleküler verilerle çalışacağız. Bu nedenle, bir sonraki bölümde bunların gösterimini ayrıntılı olarak ele almak için bu fırsatı kullanacağız.

Moleküllerin grafik olarak gösterilmesi

Kimyasal bir genel bakış olarak moleküller, kimyasal bağlarla bir arada tutulan atom grupları olarak düşünülebilir. Farklı kimyasal elementlere karşılık gelen farklı atomlar vardır; örneğin, yaygın elementler arasında karbon (C), oksijen (O), nitrojen (N) ve hidrojen (H) bulunur. Ayrıca, atomlar arasındaki bağlantıyı oluşturan farklı türde bağlar vardır, örneğin tek veya çift bağlar.

Bir molekülü, her satırın ilgili düğümün atom tipinin tek vuruşluk kodlaması olduğu bir düğüm etiket matrisi ile yönlendirilmemiş bir grafik olarak temsil edebiliriz. Ek olarak, her satırın ilişkili kenarın bağ türünün tek vuruşluk kodlaması olduğu bir kenar etiket matrisi vardır. Bu gösterimi basitleştirmek için, hidrojen atomları bazen örtük hale getirilir, çünkü konumları temel kimyasal kurallarla çıkarılabilir. Daha önce gördüğümüz kafein molekülü göz önünde bulundurulduğunda, hidrojen atomlarının örtük olduğu bir çizge gösterimi örneği *Şekil 18.4*'te gösterilmektedir:

alinpikbarbell@gmail.com



Şekil 18.4: Bir kafein molekülünün grafik gösterimi

Grafik konvolüsyonlarını anlama

Bir önceki bölümde grafik verilerinin nasıl temsil edilebileceği gösterilmiştir. Bir sonraki mantıksal adım, bu temsilleri etkili bir şekilde kullanabilecek hangi araçlara sahip olduğumuzu tartışmaktır.

Aşağıdaki alt bölümlerde, GNN'ler oluşturmak için temel bileşen olan çizge konvolüsyonlarını tanıtacağız. Bu bölümde, neden graflar üzerinde konvolüsyon kullanmak istediğimizi göreceğiz ve bu konvolüsyonların hangi özelliklere sahip olmasını istediğimizi tartışacağız. Daha sonra bir uygulama örneği üzerinden çizge konvolüsyonlarını tanıtacağız.

Grafik konvolüsyonlarının kullanılmasının ardındaki motivasyon

Grafik konvolüsyonlarını açıklamaya yardımcı olması için, *Derin* Konvolüsyonel Sinir *Ağları ile Görüntülerin Sınıflandırılması* başlıklı *14. Bölümde* ele aldığımız konvolüsyonların konvolüsyonel sinir ağlarında (CNN'ler) nasıl kullanıldığını kısaca özetleyelim. Görüntüler bağlamında, konvolüsyonu bir konvolüsyonel filtrenin bir görüntü üzerinde kaydırılması işlemi olarak düşünebiliriz; burada her adımda filtre ile alıcı alan (görüntünün o anda üzerinde bulunduğu kısmı) arasında ağırlıklı bir toplam hesaplanır.

CNN bölümünde tartışıldığı gibi, filtre belirli bir özellik için bir dedektör olarak görülebilir. Özellik tespitine yönelik bu yaklaşım, çeşitli nedenlerden dolayı görüntüler için çok uygundur, örneğin görüntü verilerine yerleştirebileceğimiz aşağıdaki öncelikler:

1. **Değişkenlik**: Bir görüntüdeki bir özelliği, bulunduğu yerden bağımsız olarak (örneğin, ötelemeden sonra) yine de tanıyabiliriz. Bir kedi, görüntünün sol üst, sağ alt veya başka bir yerinde olsa da kedi olarak tanınabilir.
2. **Lokalite**: Yakındaki pikseller yakından ilişkilidir.
3. **Hiyerarşi**: Bir görüntünün daha büyük parçaları genellikle ilişkili daha küçük parçaların kombinasyonlarına ayrılabilir. Bir kedinin başı ve bacakları vardır; başın gözleri ve burnu vardır; gözlerin göz bebekleri ve irisleri vardır.

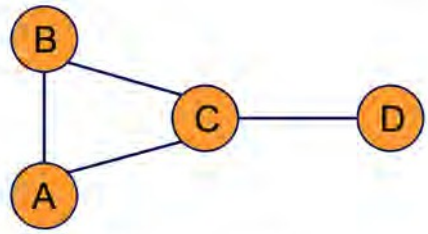
İlgilenen okuyucular, *N. Dehmamy*, *A.-L. Barabasi ve R. Yu'nun* 2019 tarihli *Understanding the Representation Power of Graph Neural Networks in Learning Graph Topology* makalesinde (https://arxiv.org/abs/1907.05008) bu önsellerin ve GNN'ler tarafından varsayılan önsellerin daha resmi bir açıklamasını bulabilirler.

Konvolüsyonların görüntüleri işlemek için çok uygun olmasının bir başka nedeni de eğitilebilir parametrelerin sayısının girdinin boyutluluğuna bağlı olmamasıdır. Örneğin 256×256 veya 9×9 görüntü üzerinde bir dizi 3×3 konvolüsyonel filtreyi eğitebilirsiniz. (Ancak, aynı görüntü farklı çözünürlüklerde sunulursa, alıcı alanlar ve dolayısıyla çıkarılan özellikler farklılık gösterecektir. Ve daha yüksek çözünürlüklü görüntüler için, daha büyük çekirdekler seçmek veya yararlı özellikleri etkili bir şekilde çıkarmak için ek katmanlar eklemek isteyebiliriz).

Görüntüler gibi grafikler de konvolüsyonel yaklaşımı haklı çıkaran doğal önceliklere sahiptir. Her iki veri türü de, görüntüler ve grafikler, yerellik önceliğini paylaşır. Ancak, yerelliği nasıl tanımladığımız farklılık gösterir. Görüntülerde öncelik 2 boyutlu uzayda yerellik iken, grafiklerde yapısal yerelliktir. Sezgisel olarak bu, bir kenar uzaklıktaki bir düğümün beş kenar uzaklıktaki bir düğüme göre ilişkili olma olasılığının daha yüksek olduğu anlamına gelir. Örneğin, bir atıf grafiğinde, bir kenar uzakta olan doğrudan atıfta bulunulan bir yayının, birden fazla ayrılık derecesine sahip bir yayına göre benzer konuya sahip olma olasılığı daha yüksektir.

Çizge verileri için katı bir öncelik **permütasyon değişmezliğidir,** yani düğümlerin sıralanması çıktıyı etkilemez. Bu durum, bir grafın düğümlerinin sıralamasının değiştirilmesinin grafın yapısını değiştirmediği *Şekil 18.5*'te gösterilmektedir:

alinpikbarbell@gmail.com



Adjacency matrix 1:

$$\begin{array}{c} D \\ B \\ A \\ C \end{array} \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Adjacency matrix 2:

$$\begin{array}{c} A \\ C \\ D \\ B \end{array} \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

Adjacency matrix 3:

$$\begin{array}{c} A \\ B \\ C \\ D \end{array} \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Şekil 18.5: Aynı grafı temsil eden farklı bitişiklik matrisleri

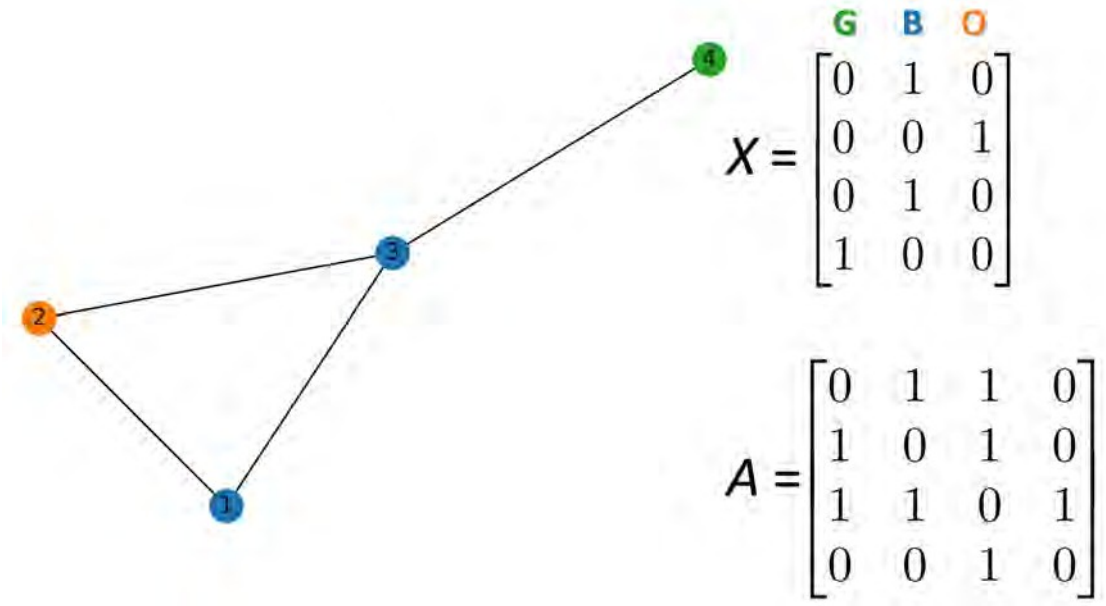
Şekil 18.5'te gösterildiği gibi, aynı çizge birden fazla bitişiklik matrisi ile temsil edilebildiğinden, sonuç olarak, herhangi bir çizge konvolüsyonunun permütasyon değişmezi olması gerekir.

Konvolüsyonel bir yaklaşım, farklı boyutlardaki grafikler için sabit bir parametre seti ile çalışabildiğinden grafikler için de arzu edilir. Bu özellik graflar için görüntülerden daha da önemlidir. Örneğin, *Bölüm 11, Sıfırdan Çok Katmanlı Yapay Sinir Ağı Uygulaması*'nda gördüğümüz gibi, tam bağlantılı bir yaklaşımın (örneğin, çok katmanlı *bir* algılayıcı kullanarak) mümkün olabileceği sabit bir çözünürlüğe sahip birçok görüntü veri kümesi vardır. Buna karşın, çoğu grafik veri kümesi farklı boyutlarda grafikler içerir.

Görüntü konvolüsyon operatörleri standartlaştırılmış olsa da, birçok farklı türde çizge konvolüsyonu vardır ve yeni çizge konvolüsyonlarının geliştirilmesi çok aktif bir araştırma alanıdır. Bizim odak noktamız, okuyucuların kullanmak istedikleri GNN'ler hakkında mantık yürütebilmeleri için genel fikirler sunmaktır. Bu amaçla, aşağıdaki alt bölümde PyTorch'ta temel bir çizge konvolüsyonunun nasıl uygulanacağı gösterilecektir. Daha sonra, bir sonraki bölümde, PyTorch'ta sıfırdan basit bir GNN inşa edeceğiz.

Temel bir grafik konvolüsyonunun uygulanması

Bu alt bölümde, temel bir grafik konvolüsyon fonksiyonunu tanıtacağız ve bir grafiğe uygulandığında ne olduğunu göreceğiz. Aşağıdaki grafiği ve gösterimini ele alalım:



Şekil 18.6: Bir grafiğin gösterimi

Şekil 18.6, $n \times n$ bitişiklik matrisi $\boldsymbol{A}$ ve $n \times f_{in}$ düğüm özellik matrisi $\boldsymbol{X}$ tarafından belirtilen düğüm etiketlerine sahip yönlendirilmemiş bir grafiği göstermektedir; burada tek özellik, her düğümün renginin tek vuruşluk bir gösterimidir - yeşil (G), mavi (B) veya turuncu (O).

Grafik manipülasyonu ve görselleştirmesi için en çok yönlü kütüphanelerden biri, bir etiket matrisi $\boldsymbol{X}$ ve bir düğüm matrisi $\boldsymbol{A}'$dan grafiklerin nasıl oluşturulacağını göstermek için kullanacağımız NetworkX'tir.

NetworkX'i Yükleme

NetworkX, grafikleri işlemek ve görselleştirmek için kullanışlı bir Python kütüphanesidir. Pip aracılığıyla kurulabilir:

```
pip install networkx
```

Bu bölümdeki grafik görselleştirmelerini oluşturmak için 2.6.2 sürümünü kullandık. Daha fazla bilgi için lütfen https://networkx.org adresindeki resmi web sitesini ziyaret edin.

NetworkX kullanarak *Şekil 18.6*'da gösterilen grafiği aşağıdaki gibi oluşturabiliriz:

```
>> import numpy as np
>> import networkx as nx
```

 ilinpikbarbell@gmail.com

```
>>> G = nx.Graph()
... # Grafik çizersek renkler için onaltılı kodlar
>>> mavi, turuncu, yeşil = "#1f77b4", " #ff7f0e", "#2ca02c"
>>> G.add_nodes_from([
...         (1, {"renk": mavi}),
...         (2, {"renk": turuncu}),
...         (3, {"renk": mavi}),
...         (4, {"renk": yeşil})
... ])
>>> G.add_edges_from([(1,2), (2,3), (1,3), (3,4)])
> > >   A = np.asarray(nx.adjacency_matrix(G).todense())
>> print(A)
[[0 1 1 0]
[1 0 1 0]
[1 1 0 1]
[0 0 1 0]]

>> def build_graph_color_label_representation(G, mapping_dict):
...         one_hot_idxs = np.array([mapping_dict[v] for v in
...                 nx.get_node_attributes(G, 'color').values()])
>>>         one_hot_encoding = np.zeros(
...                 (one_hot_idxs.size, len(mapping_dict))
>>>         one_hot_encoding[
...                 np.arange(one_hot_idxs.size), one_hot_idxs] = 1
>>>         return one_hot_encoding
> >   X = build_graph_color_label_representation(
...         G, {yeşil: 0, mavi: 1, turuncu: 2})
>> print(X) [[0.,
1., 0.],
[0., 0., 1.],
[0., 1., 0.],
[1., 0., 0.]]
```

Önceki kodda oluşturulan grafiği çizmek için aşağıdaki kodu kullanabiliriz:

```
>>> color_map = nx.get_node_attributes(G, 'color').values()
>> nx.draw(G,with_labels=True, node_color=color_map)
```

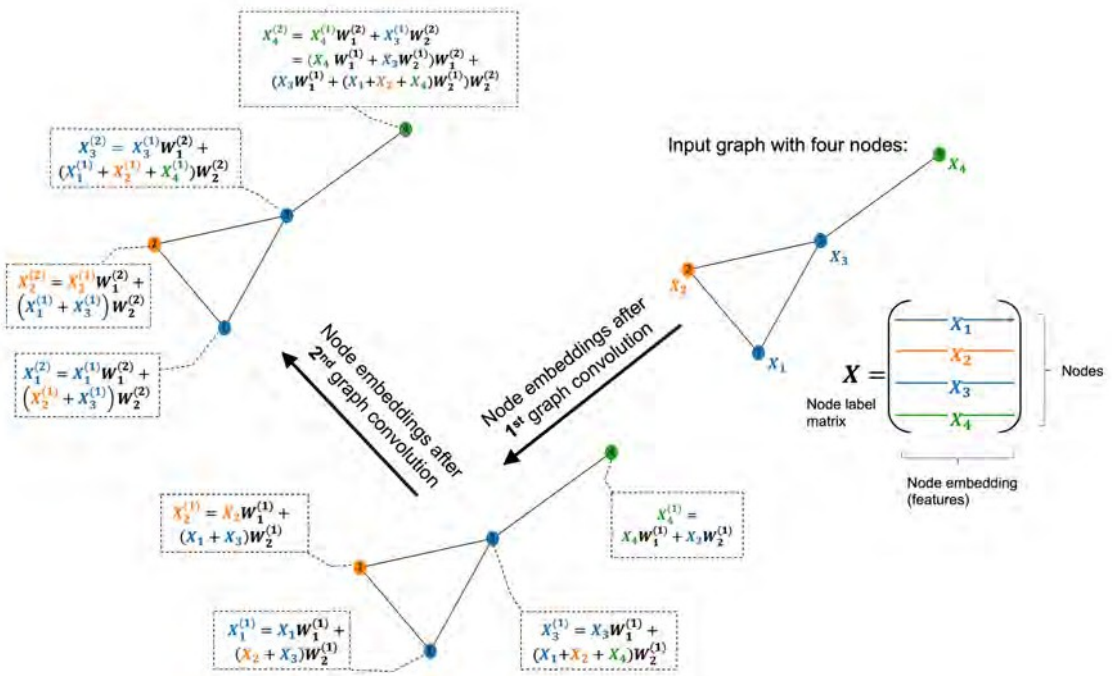
Yukarıdaki kod örneğinde, ilk olarak NetworkX'ten yeni bir Graph nesnesi başlattık. Daha sonra görselleştirme için renk özellikleriyle birlikte 1'den 4'e kadar düğümler ekledik. Düğümleri ekledikten sonra bağlantılarını (kenarlarını) belirledik. NetworkX'in adjacency_matrix kurucusunu kullanarak ***A*** ***bitişiklik*** matrisini oluşturduk ve özel build_graph_color_label_representation fonksiyonumuz, Graph nesnesine daha önce eklediğimiz bilgilerden ***X*** düğüm etiketi matrisini oluşturdu.

Çizge konvolüsyonları ile $\boldsymbol{X'}$in her bir satırını, o satıra karşılık gelen düğümde depolanan bilginin gömülmesi olarak yorumlayabiliriz. Çizge konvolüsyonları, komşularının ve kendilerinin gömülmelerine dayalı olarak her düğümdeki gömülmeleri günceller. Örnek uygulamamız için, çizge konvolüsyonu aşağıdaki biçimi alacaktır:

$$\boldsymbol{x}'_i = \boldsymbol{x}_i \boldsymbol{W}_1 + \sum_{j \in \mathcal{N}(i)} \boldsymbol{x}_j \boldsymbol{W}_2 + b$$

Burada, $\boldsymbol{x}'_i$ i düğümü için güncellenmiş gömme; $\boldsymbol{W}_1$ ve $\boldsymbol{W}_2$ öğrenilebilir filtre ağırlıklarının $f_{in} \times f_{out}$ matrisleridir; ve b f_{out} uzunluğunda öğrenilebilir bir yanlılık vektörüdür.

İki ağırlık matrisi $\boldsymbol{W}_1$ ve $\boldsymbol{W}_2$, her sütunun ayrı bir filtre olduğu filtre bankaları olarak düşünülebilir. Bu filtre tasarımının, grafik verilerindeki yerellik önceliği geçerli olduğunda en etkili olduğunu unutmayın. Bir düğümdeki bir değer, birçok kenar ötedeki başka bir düğümdeki değerle yüksek oranda ilişkiliyse, tek bir konvolüsyon bu ilişkiyi yakalayamayacaktır. *Şekil 18.7*'de gösterildiği gibi konvolüsyonların istiflenmesi daha uzak ilişkileri yakalayacaktır (basitlik için önyargıyı sıfıra ayarladık):



Şekil 18.7: Bir grafikten ilişkileri yakalama

 olimpikbarbell@gmail.com

Şekil 18.7'de gösterilen grafik konvolüsyonunun tasarımı, grafik verilerine ilişkin öncüllerimize uymaktadır, ancak komşular üzerindeki toplamın matris biçiminde nasıl uygulanacağı açık olmayabilir. Burada bitişiklik matrisi ***A'yı*** kullanırız. Bu konvolüsyonun matris formu $\boldsymbol{XW}_1 + \boldsymbol{AXW}_2$ şeklindedir. Burada, 1'ler ve 0'lardan oluşan bitişiklik matrisi, düğümleri seçmek ve istenen toplamları hesaplamak için bir maske görevi görür. NumPy'de, bu katmanın başlatılması ve önceki

```
>>> f_in, f_out = X.shape[1], 6
>>> W_1 = np.random.rand(f_in, f_out)
>>> W_2 = np.random.rand(f_in, f_out)
>>> h = np.dot(X, W_1)+ np.dot(np.dot(A,X), W_2)
```

grafik üzerinde bir ileri geçişin hesaplanması aşağıdaki gibi yazılabilir:

Bir grafik konvolüsyonunun ileri geçişini hesaplamak bu kadar kolaydır.

Nihayetinde, bir çizge evrişim katmanının ***A*** tarafından sağlanan yapısal (bağlantı) bilgiyi kullanarak ***X'te*** kodlanan düğüm bilgisinin temsilini güncellemesini istiyoruz. Bunu yapmanın birçok potansiyel yolu vardır ve bu, geliştirilen çok sayıda çizge evrişim türünde ortaya çıkar.

Farklı grafik konvolüsyonları hakkında konuşmak için, genel olarak, birleştirici bir çerçeveye sahip olmaları güzel olurdu. Neyse ki, böyle bir çerçeve *Justin Gilmer* ve meslektaşları tarafından *Kuantum Kimyası için Nöral Mesaj Geçişi'nde* sunuldu, 2017, https://arxiv.org/abs/1704.01212.

Bu **mesaj geçirme** çerçevesinde, grafikteki her düğümün ilişkili bir gizli durumu vardır $h_i^{(t)}$, burada *i* düğümün *t* zaman adımındaki indeksidir. $h_i^{(0)}$ başlangıç değeri $\boldsymbol{X}_i$ olarak tanımlanır ve ***X***'in ilişkili satırıdır.
i düğümü ile.

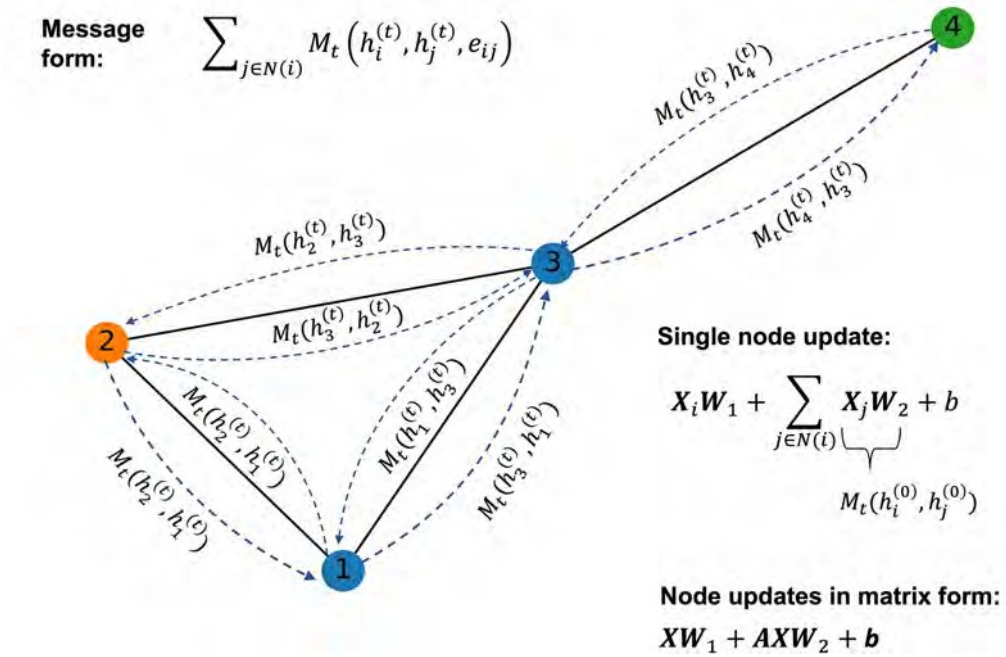
Her bir çizge konvolüsyonu bir mesaj iletme aşamasına ve bir düğüm güncelleme aşamasına ayrılabilir. *N(i), i düğümünün* komşuları olsun. Yönlendirilmemiş graflar için *N(i), i düğümü* ile bir kenarı paylaşan düğümlerin kümesidir. Yönlendirilmiş graflar için *N(i),* uç noktası *i* düğümü olan bir kenara sahip düğümlerin kümesidir. Mesaj geçirme aşaması aşağıdaki gibi formüle edilebilir:

$$m_i^{(t)} = \sum_{j \in N(i)} M_t(h_i^{(t)}, h_j^{(t)}, e_{ij})$$

Burada, M_t bir mesaj fonksiyonudur. Örnek katmanımızda bu mesaj fonksiyonunu $M_t = h_j^{(t)} W_2 W$ olarak tanımlıyoruz.

U_t güncelleme fonksiyonu ile düğüm güncelleme aşaması $h_i^{(t+1)} = U_t(h_i^{(t)}, m_i^{(t)})$ şeklindedir. Örnek katmanımızda bu güncelleme $h_i^{(t+1)} = h_i^{(t)} W_1 W + m_i + b$ şeklindedir.

Şekil 18.8 mesaj geçirme fikrini görselleştirmekte ve uyguladığımız konvolüsyonu özetlemektedir:



Şekil 18.8: Grafik ve mesaj formu üzerinde uygulanan konvolüsyonlar

Bir sonraki bölümde, bu grafik konvolüsyon katmanını PyTorch'ta uygulanan bir GNN modeline dahil edeceğiz.

PyTorch'ta sıfırdan bir GNN uygulama

Bir önceki bölümde grafik konvolüsyon işleminin anlaşılması ve uygulanmasına odaklanılmıştı. Bu bölümde, sıfırdan başlarsanız bu yöntemleri grafiklere nasıl uygulayacağınızı göstermek için bir grafik sinir ağının temel bir uygulamasında size yol göstereceğiz. Bu yaklaşım karmaşık görünüyorsa endişelenmeyin; GNN'ler uygulanması nispeten karmaşık modellerdir. Bu nedenle, daha sonraki bir bölümde, çizge sinir ağlarının uygulanmasını ve veri yönetimini kolaylaştırmak için araçlar sağlayan PyTorch Geometric'i tanıtacağız.

 alinpikbarbell@gmail.com

NodeNetwork modelinin tanımlanması

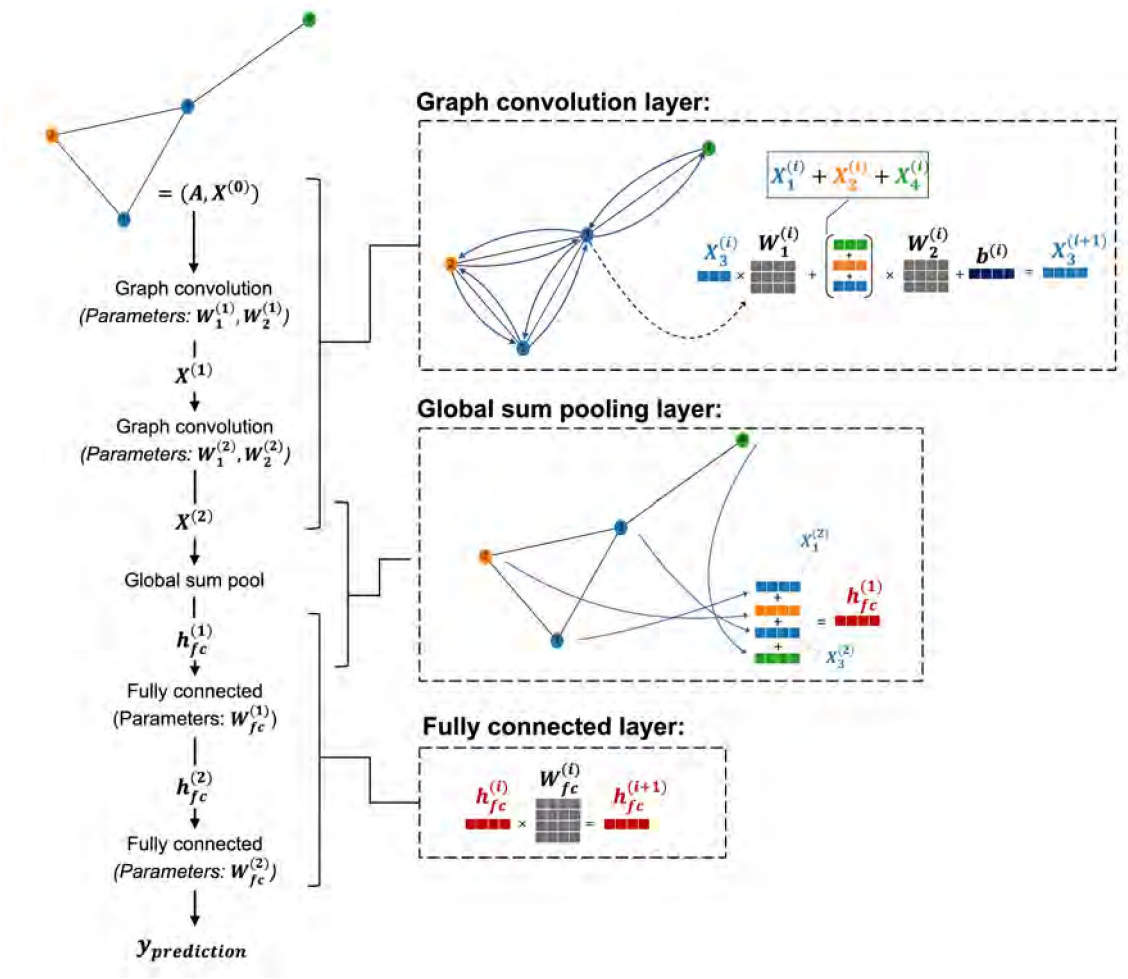
Bu bölüme bir GNN'nin PyTorch tarafından sıfırdan uygulanmasını göstererek başlayacağız. NodeNetwork adını verdiğimiz ana sinir ağı modeliyle başlayarak yukarıdan aşağıya bir yaklaşım izleyeceğiz ve ardından bireysel ayrıntıları dolduracağız:

```
import networkx as nx
import torch
from torch.nn.parameter import Parametre
import numpy as np
import math
import torch.nn.functional as F

class NodeNetwork(torch.nn.Module):
    def init (self, input_features):
        super(). init ()
        self.conv_1 = BasicGraphConvolutionLayer (
            input_features, 32)
        self.conv_2 = BasicGraphConvolutionLayer(32, 32)
        self.fc_1 = torch.nn.Linear(32, 16) self.out_layer =
        t o r c h . nn.Linear(16, 2)

    def forward(self, X, A, batch_mat): x
        = F .relu(self.conv_1(X, A ) )  x =
        F.relu(self.conv_2(x, A))
        output = global_sum_pool(x, batch_mat)
        output = self.fc_1(output)
        çıktı = self.out_layer(çıktı) return
        F.softmax(çıktı, dim=1)
```

Az önce tanımladığımız NodeNetwork modeli aşağıdaki gibi özetlenebilir:

1. İki grafik konvolüsyonu gerçekleştirin (self.conv_1 ve self.conv_2)
2. Daha sonra tanımlayacağımız global_sum_pool aracılığıyla tüm düğüm katıştırmalarını havuzlayın
3. Havuzlanmış katıştırmaları iki tam bağlı katman (self.fc_1 ve self.out_ katmanı) üzerinden çalıştırın
4. Softmax aracılığıyla bir sınıf-üyelik olasılığı çıktısı

Ağın yapısı, her katmanın ne yaptığının görselleştirilmesiyle birlikte şu şekilde özetlenmiştir *Şekil 18.9*:



Şekil 18.9: Her bir sinir ağı katmanının görselleştirilmesi

Grafik konvolüsyon katmanları ve küresel havuzlama gibi münferit hususlar sonraki alt bölümlerde ele alınacaktır.

NodeNetwork'ün grafik konvolüsyon katmanının kodlanması

Şimdi, önceki NodeNetwork sınıfının içinde kullanılan grafik konvolüsyon işlemini (BasicGraphConvolutionLayer) tanımlayalım:

```
class BasicGraphConvolutionLayer(torch.nn.Module):
    def init (self, in_channels, out_channels):
        super(). init ()
        self.in_channels = in_channels
```

 ilinpikbarbell@gmail.com

```
        self.out_channels = out_channels self.W2
        = Parametre(torch.rand(
            (in_channels, out_channels), dtype=torch.float32))
        self.W1 = Parametre(torch.rand(
            (in_channels, out_channels), dtype=torch.float32))

        self.bias = Parametre(torch.zeros(
                out_channels, dtype=torch.float32))
    def ileri(self, X, A):
        potential_msgs = torch.mm(X, self.W2) propagated_msgs
        = torch.mm(A, potential_msgs) root_update =
        t o r c h . m m ( X, self.W1)
        output = propagated_msgs + root_update + self.bias
        return output
```

Tam bağlantılı katmanlar ve görüntü evrişimli katmanlarda olduğu gibi, katman çıktılarının doğrusal kombinasyonunun kesişiminin (ReLU gibi doğrusal olmayan bir uygulamanın öncesinde) değişebilmesi için bir sapma terimi ekliyoruz. forward() yöntemi, bir önceki alt bölümde tartıştığımız ileri geçişin matris formunu bir önyargı terimi ekleyerek uygular.

BasicGraphConvolutionLayer'ı denemek için, daha önce *temel bir* grafik *konvolüsyonu uygulama* bölümünde tanımladığımız *grafiğe* ve bitişiklik matrisine uygulayalım:

```
>> print('X.shape:', X.shape)
X.shape: (4, 3)

>> print('A.shape:', A.shape)
A.shape: (4, 4)

>> basiclayer = BasicGraphConvolutionLayer(3, 8)
>> out = basiclayer(
...         X=torch.tensor(X, dtype=torch.float32),
...         A=torch.tensor(A, dtype=torch.float32)
... )

>> print('Çıktı şekli:', out.shape) Çıktı şekli:
torch.Size([4, 8])
```

Yukarıdaki kod örneğine dayanarak, BasicGraphConvolutionLayer'ımızın üç özellikten oluşan dört düğümlü grafiği sekiz özellikli bir temsile dönüştürdüğünü görebiliriz.

Değişken grafik boyutlarıyla başa çıkmak için küresel bir havuzlama katmanı ekleme

Daha sonra, NodeNetwork sınıfında kullanılan global_sum_pool() fonksiyonunu tanımlıyoruz, burada global_sum_pool() global bir havuzlama katmanı uyguluyor. Küresel havuzlama katmanları, bir grafın tüm düğüm katıştırmalarını sabit boyutlu bir çıktıda toplar. *Şekil 18.9*'da gösterildiği gibi, global_sum_pool() bir grafın tüm düğüm katıştırmalarını toplar. Bu global havuzlamanın, *Derin Evrişimli Sinir Ağları ile Görüntülerin Sınıflandırılması* başlıklı *14. Bölümde* gördüğümüz gibi, veriler tam bağlı katmanlardan geçirilmeden önce kullanılan CNN'lerde kullanılan global ortalama havuzlamaya nispeten benzediğini belirtmek isteriz.

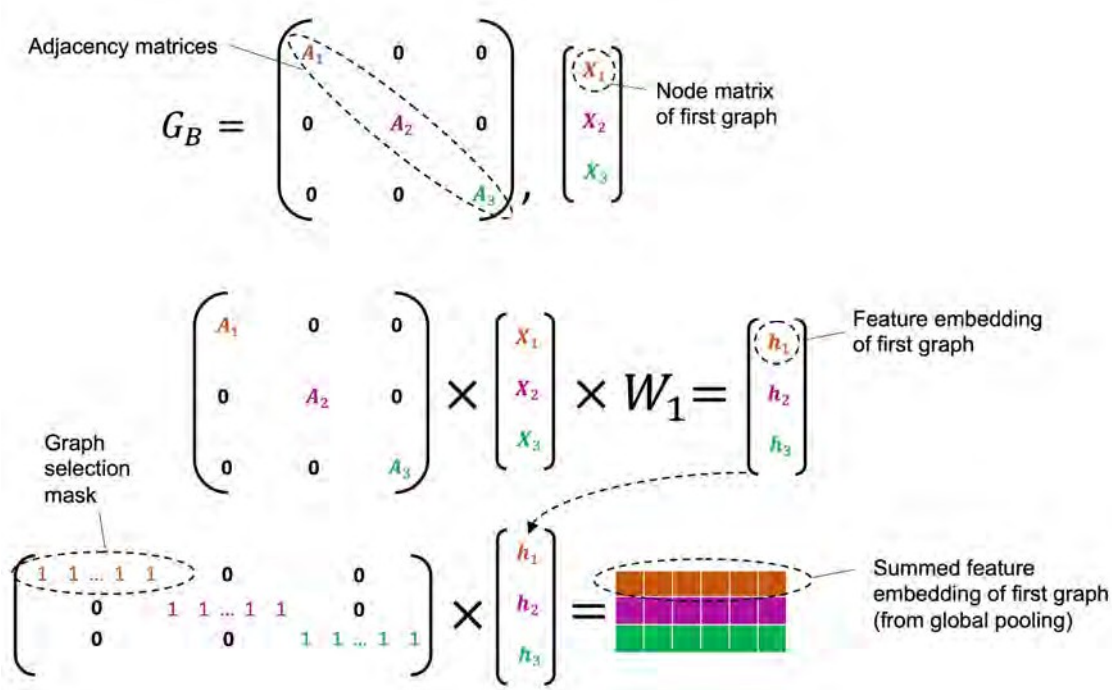
Tüm düğüm katıştırmalarının toplanması bilgi kaybına neden olur, bu nedenle verilerin yeniden şekillendirilmesi tercih edilir, ancak graflar farklı boyutlarda olabileceğinden bu mümkün değildir. Global havuzlama, toplam, maksimum ve ortalama gibi herhangi bir permütasyon değişmez fonksiyonu ile yapılabilir. İşte global_sum_pool() uygulaması:

```
def global_sum_pool(X, batch_mat):
    if batch_mat is None or batch_mat.dim() == 1:
        return torch.sum(X, dim=0).unsqueeze(0)
    Başka:
        return torch.mm(batch_mat, X)
```

Veriler gruplanmamışsa veya grup boyutu bir ise, bu fonksiyon sadece mevcut düğüm katıştırmalarını toplar. Aksi takdirde, katıştırmalar, grafik verilerinin nasıl katıştırıldığına dayalı bir yapıya sahip olan batch_mat ile çarpılır.

Bir veri kümesindeki tüm veriler aynı boyutluluğa sahip olduğunda, verileri toplu hale getirmek, verileri istifleyerek bir boyut eklemek kadar basittir. (Yan not: PyTorch'taki varsayılan batching işlevinde çağrılan fonksiyona tam anlamıyla stack denir). Grafik boyutları değişkenlik gösterdiğinden, dolgu kullanılmadığı sürece bu yaklaşım grafik verileriyle uygulanabilir değildir. Bununla birlikte, grafik boyutlarının önemli ölçüde değişebildiği durumlarda dolgu verimsiz olabilir. Genellikle, değişen çizge boyutlarıyla başa çıkmanın daha iyi yolu, her bir yığını, yığındaki her bir çizgenin diğerlerinden bağlantısı kesilmiş bir alt çizge olduğu tek bir çizge olarak ele almaktır. Bu durum *Şekil 18.10*'da gösterilmektedir:

ilinpikbarbell@gmail.com



Şekil 18.10: Değişken grafik boyutları ile nasıl başa çıkılır?

*Şekil 18.10'*u daha resmi olarak tanımlamak için, bize n_1 , ..., n_k boyutlarında G_1 , ..., G_k grafları verildiğini varsayalım.

f düğüm başına özellik. Buna ek olarak, ilgili bitişiklik matrisleri $\boldsymbol{A}_1$, ..., $\boldsymbol{A}_k$ ve $\boldsymbol{X}_1$, ..., $\boldsymbol{X}_k$ özellik matrisleri olsun. N toplam düğüm sayısı, N $N\sum^{kk} n_i$, $s_{1i\bar{i}}$ 0 ve $s_i = s_{i-1} + n_{i-1}$ için $1 < i\ ik$. Şekilde gösterildiği gibi, N×N bitişiklik matrisi $\boldsymbol{A}_B$ ve N×f özellik matrisi $\boldsymbol{X}_B$ ile bir G_B grafiği tanımlıyoruz. Python indeks gösterimini kullanarak, $\boldsymbol{A}_B\ [s_i : s_i + n_i\ ,\ s_i + n_i\ ] = \boldsymbol{A}_i$ ve $\boldsymbol{A}_B$ 'nin bu indeks kümeleri dışındaki diğer tüm elemanları 0'dır. Ek olarak, $\boldsymbol{X}_B\ [s_i : s_i + n_i\ , :] = \boldsymbol{X}_i$.

Tasarım gereği, bağlantısı kesilmiş düğümler hiçbir zaman bir grafik konvolüsyonunun aynı alıcı alanında olmayacaktır. Sonuç olarak, G_B gradyanlarını grafik konvolüsyonları aracılığıyla geriye doğru yayarken, yığındaki her grafiğe bağlı gradyanlar bağımsız olacaktır. Bu, bir dizi grafik konvolüsyonunu bir f fonksiyonu olarak ele alırsak, $h_B = f(X_B\ ,\ A_B\ )$ ve $h_i = f(X_i\ ,\ A_i\ )$ ise, $h_B\ [s_i : s_i + n,\ :] = h_i$ anlamına gelir. Toplam global havuzlama her bir h_i 'nin toplamını h_B 'den ayrı vektörler olarak çıkarırsa, bu vektör yığınını tam bağlı katmanlardan geçirmek, yığındaki her bir öğenin gradyanlarını tüm geri yayılım boyunca ayrı tutacaktır.

Bu, global_sum_pool()içindeki batch_mat'in amacıdır - yığındaki grafikleri ayrı tutan bir grafik seçim maskesi olarak hizmet etmek. Bu maskeyi n_1 , ..., n_k boyutlarındaki grafikler için aşağıdaki kodla oluşturabiliriz:

```
def get_batch_tensor(graph_sizes):
    starts = [sum(graph_sizes[:idx])
                for idx in range(len(graph_sizes))]
    stops = [starts[idx] + graph_sizes[idx]
              for idx in range(len(graph_sizes))]
    tot_len = sum(graph_sizes)
    batch_size = len(graph_sizes)
    batch_mat = torch.zeros([batch_size, tot_len]).float()
    for idx, starts_and_stops in enumerate(zip(starts, stops)):
        start = s t a r t s _ a n d _ s t o p s [0]
        stop = starts_and_stops[1]
        batch_mat[idx,start:stop] = 1
    return batch_mat
```

Böylece, bir yığın boyutu, b, verildiğinde, batch_mat, batch_mat[i-1, s_i :$s_i + n_i$] = 1 $\leq i \leq k$ için 1 ve bu indeks kümelerinin dışındaki elemanların 0 olduğu bir $b \times N$ matrisidir. Aşağıda, bazı G_B ve karşılık gelen bir yığın matrisinin bir gösterimini oluşturmak için bir harmanlama işlevi verilmiştir:

```
# batch, her biri # bir grafiğin temsilini ve etiketini
içeren sözlüklerin bir listesidir
def collate_graphs(batch):
    adj_mats = [graph['A'] for graph in batch]
    sizes = [A.size(0) for A in adj_mats] tot_size
    = sum(sizes)
    # toplu matris oluşturun
    batch_mat = get_batch_tensor(sizes)
    # özellik matrislerini birleştirin
    feat_mats = torch.cat([graph['X'] for graph in batch], dim=0)
    # etiketleri birleştir
    labels = torch.cat([graph['y'] for graph in batch], dim=0)
    # bitişiklik matrislerini birleştir
    batch_adj = torch.zeros([tot_size, tot_size], dtype=torch.float32)
    accum = 0
    adj_mats içindeki adj için:
        g_size = adj.shape[0]
        batch_adj[accum:accum+g_size,accum:accum+g_size] = adj accum
        = accum + g_size
    repr_and_label = {'A': batch_adj,
```

 olimpikbarbell@gmail.com

```
                'X': feat_mats, 'y': labels,
                'batch': batch_mat}
        return repr_and_label
```

DataLoader'ı Hazırlama

Bu bölümde, önceki alt bölümlerdeki kodların nasıl bir araya geldiğini göreceğiz. İlk olarak, bazı grafikler oluşturacağız ve bunları bir PyTorch Veri Kümesine koyacağız. Ardından, collate fonksiyonumuzu GNN'miz için bir DataLoader'da kullanacağız.

Ancak grafikleri tanımlamadan önce, daha sonra kullanacağımız bir sözlük temsili oluşturan bir işlev uygulayalım:

```
def get_graph_dict(G, mapping_dict):
    # Fonksiyon G grafiğinin sözlük temsilini oluşturur
    A = torch.from_numpy(
        np.asarray(nx.adjacency_matrix(G).todense()).float()
    # build_graph_color_label_representation()
    # ilk örnek grafik ile tanıtıldı
    X = torch.from_numpy(
      build_graph_color_label_representation(
                G, mapping_dict)).float()
    # bu örnek için belirli bir görev olmadığından kludge
    y = torch.tensor([[1,0]]).float()
    return {'A': A, 'X': X, 'y': y, 'batch': Yok}
```

Bu fonksiyon bir NetworkX grafiğini alır ve bitişiklik matrisi A, düğüm özellik matrisi X ve ikili etiket y'yi içeren bir sözlük döndürür. Bu modeli gerçek dünyadaki bir görev üzerinde eğitmeyeceğimiz için etiketleri keyfi olarak ayarlıyoruz. Ardından, nx.adjacency_matrix() bir NetworkX grafiğini alır ve todense() kullanarak yoğun bir np.array formuna dönüştürdüğümüz seyrek bir temsil döndürür.

Şimdi grafikleri oluşturacağız ve NetworkX grafiklerini ağımızın işleyebileceği bir biçime dönüştürmek için get_graph_dict işlevini kullanacağız:

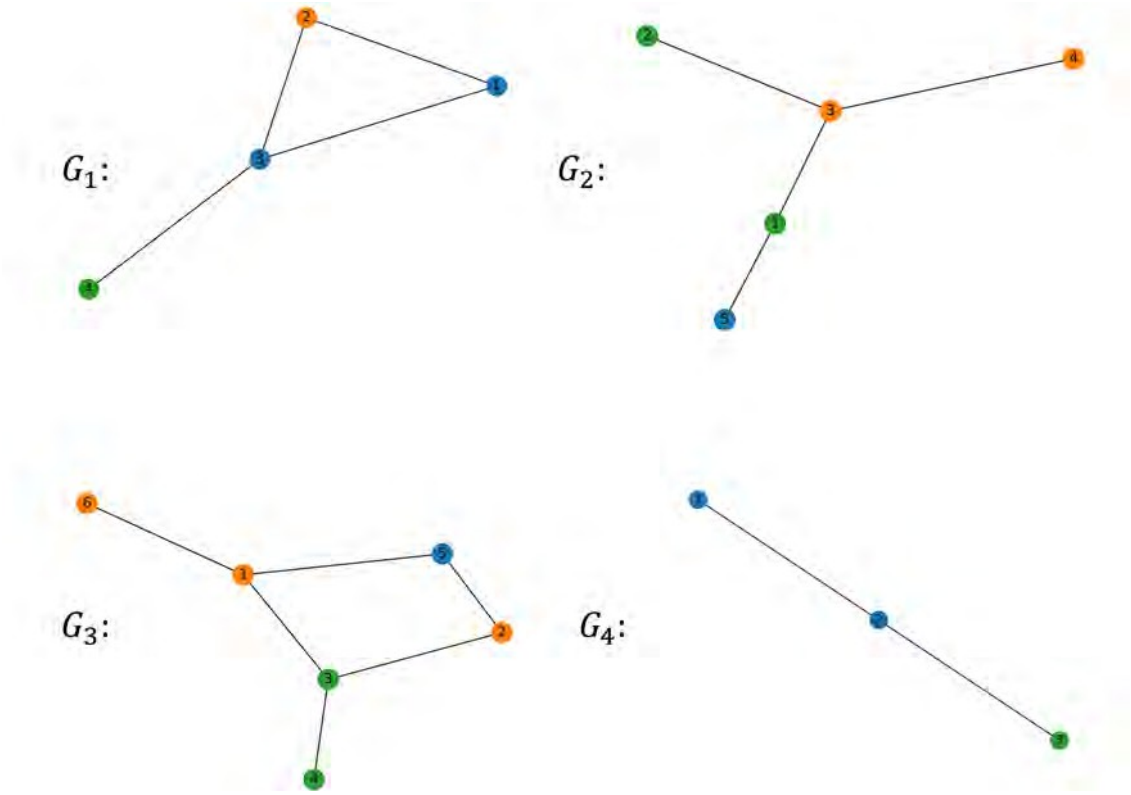
```
>> # veri kümesi olarak ele almak için 4 grafik oluşturma
>>> mavi, turuncu, yeşil = "#1f77b4", "#ff7f0e", "#2ca02c"
>>> mapping_dict= {green:0, blue:1, orange:2}
>>> G1 = nx.Graph()
>>> G1.add_nodes_from([
...     (1,{"renk": mavi}),
...     (2,{"renk": turuncu}),
...     (3,{"renk": mavi}),
...     (4,{"renk": yeşil})
... ])
```

6587191CA584E

```
>>> G1.add_edges_from([(1, 2), (2, 3), (1, 3), (3, 4)])
>>> G2 = nx.Graph()
>>> G2.add_nodes_from([
...         (1,{"renk": yeşil}),
...         (2,{"renk": yeşil}),
...         (3,{"renk": turuncu}),
...         (4,{"renk": turuncu}),
...         (5,{"renk": mavi})
... ])
>>> G2.add_edges_from([(2, 3),(3, 4),(3, 1),(5, 1)])
>>> G3 = nx.Graph()
>>> G3.add_nodes_from([
...         (1,{"renk": turuncu}),
...         (2,{"renk": turuncu}),
...         (3,{"renk": yeşil}),
...         (4,{"renk": yeşil}),
...         (5,{"renk": mavi}),
...         (6,{"color":orange})
... ])
>> G3.add_edges_from([(2,3), (3,4), (3,1), (5,1), (2,5), (6,1)])
>>> G4 = nx.Graph()
>>> G4.add_nodes_from([
...         (1,{"renk": mavi}),
...         (2,{"renk": mavi}),
...         (3,{"renk": yeşil})
... ])
>>> G4.add_edges_from([(1, 2), (2, 3)])
>>> graph_list = [get_graph_dict(graph, mapping_dict) for graph in
...         [G1, G2, G3, G4]]
```

 alinpikbarbell@gmail.com

Bu kodun oluşturduğu grafikler *Şekil 18.11*'de görselleştirilmiştir:



Şekil 18.11: Oluşturulan dört grafik

Bu kod bloğu dört NetworkX grafiği oluşturur ve bunları bir listede saklar. Burada, nx.Graph() yapıcısı boş bir grafiği başlatır ve add_nodes_from() bir tuple listesinden boş grafiğe düğümler ekler. Her tuple'daki ilk öğe düğümün adıdır ve ikinci öğe o düğümün niteliklerinin bir sözlüğüdür.

Bir grafın add_edges_from() metodu, her bir tuple'ın elemanları (düğümler) arasında bir kenar tanımladığı bir tuple listesi alır. Şimdi, bu grafikler için bir PyTorch Veri Kümesi oluşturabiliriz:

```
from torch.utils.data import Dataset
class ExampleDataset(Dataset):
    # Grafik listemizi kullanacak basit PyTorch veri kümesi
    def init (self, graph_list):
        self.graphs = graph_list
    def len (self):
        return len(self.graphs)

    def getitem (self,idx): mol_rep =
        self.graphs[idx] return
        mol_rep
```

Özel bir Veri Kümesi kullanmak gereksiz bir çaba gibi görünse de, collate_ graphs() işlevinin bir DataLoader'da nasıl kullanılabileceğini göstermemizi sağlar:

```
>> from torch.utils.data import DataLoader
>> dset = ExampleDataset(graph_list)
>> # Özel collate fonksiyonumuzu nasıl kullandığımıza dikkat edin
>> loader = DataLoader(
...         dset, batch_size=2, shuffle=False,
...         collate_fn=collate_graphs)
```

Tahmin yapmak için NodeNetwork'ü kullanma

Gerekli tüm fonksiyonları tanımladıktan ve DataLoader'ı kurduktan sonra, şimdi yeni bir NodeNetwork ve grafik verilerimize uygulayın:

```
>> node_features = 3
>> net = NodeNetwork(node_features)

>>> batch_results = []
>> for b in loader:
...         batch_results.append(
...             net(b['X'], b ['A'], b['batch']).detach())
```

Kısa olması için bir eğitim döngüsü eklemedik; ancak GNN modeli, tahmin edilen ve gerçek sınıf etiketleri arasındaki kaybı hesaplayarak, kaybı .backward() aracılığıyla geriye doğru yayarak ve model ağırlıklarını gradyan iniş tabanlı bir optimize edici aracılığıyla güncelleyerek normal bir şekilde eğitilebilir. Bunu okuyucu için isteğe bağlı bir alıştırma olarak bırakıyoruz. Bir sonraki bölümde, daha sofistike GNN kodu uygulayan PyTorch Geometric'ten bir GNN uygulaması ile bunun nasıl yapılacağını göstereceğiz.

 ilinpikbarbell@gmail.com

Önceki kodumuza devam etmek için, şimdi DataLoader olmadan doğrudan modele tek bir giriş grafiği sağlayalım:

```
>>> G1_rep = dset[1]
>> G1_single = net(
...         G1_rep['X'], G1_rep['A'], G1_rep['batch']).detach()
```

Şimdi GNN'yi tek bir grafiğe (G1_single) ve DataLoader'daki ilk grafiğe (ayrıca shuffle=False olarak ayarladığımız için garanti ettiğimiz ilk grafik olan G1) uygulayarak toplu yükleyicinin doğru çalışıp çalışmadığını iki kez kontrol edebiliriz. torch.isclose() (yuvarlama hatalarını hesaba katmak için) kullanarak görebileceğimiz gibi, sonuçlar umduğumuz gibi eşdeğerdir:

```
>>>  G1_batch = batch_results[0][1]
>>> torch.all(torch.isclose(G1_single, G1_batch))
tensor(True)
```

Tebrikler! Artık temel bir GNN'yi nasıl oluşturacağınızı, kuracağınızı ve çalıştıracağınızı anladınız. Ancak, bu girişten sonra, muhtemelen grafik verilerini yönetmenin ve manipüle etmenin biraz zahmetli olabileceğini fark etmişsinizdir. Ayrıca, kenar etiketlerini kullanan bir grafik konvolüsyonu bile oluşturmadık, bu da sorunları daha da karmaşık hale getirecektir. Neyse ki, birçok GNN katmanının uygulamalarını sağlayarak bunu çok daha kolay hale getiren bir paket olan PyTorch Geometric var. Bu kütüphaneyi, bir sonraki alt bölümde molekül verileri üzerinde daha karmaşık bir GNN'nin uygulanması ve eğitilmesine ilişkin uçtan uca bir örnekle tanıtacağız.

PyTorch Geometrik kütüphanesini kullanarak bir GNN uygulama

Bu bölümde, GNN'lerin eğitim sürecini basitleştiren PyTorch Geometric kütüphanesini kullanarak bir GNN uygulayacağız. GNN'yi küçük moleküllerden oluşan bir veri kümesi olan QM9'a, bir molekülün yükünün bir elektrik alanı tarafından bozulma eğiliminin bir ölçüsü olan izotropik polarize edilebilirliği önceden belirlemek için uyguluyoruz.

PyTorch Geometric'i Yükleme

PyTorch Geometric conda veya pip aracılığıyla kurulabilir. İşletim sisteminiz için önerilen kurulum komutunu seçmek için https://pytorch-geometric.readthedocs.io/en/latest/ notes/installation.html adresindeki resmi dokümantasyon web sitesini ziyaret etmenizi öneririz. Bu bölümde, torch-scatter ve torch-sparse bağımlılıkları ile birlikte 2.0.2 sürümünü yüklemek için pip kullandık:

```
pip install torch-scatter==2.0.9
pip install torch-sparse==0.6.12
pip install torch-geometric==2.0.2
```

Küçük moleküllerden oluşan bir veri kümesi yükleyerek başlayalım ve PyTorch Geometric'in verileri nasıl sakladığına bakalım:

```
>> # Bu bölümdeki tüm örnekler için aşağıdaki içe aktarımları kullanacağız.
>> # torch_geometric'in DataLoader'ını kullandığımızı unutmayın.
>> import torch
> >  from torch_geometric.datasets import QM9
>> from torch_geometric.loader import DataLoader
>> from torch_geometric.nn import NNConv, global_add_pool
>> import torch.nn.functional as F
>> import torch.nn as nn
>> import numpy as np
>> # QM9 küçük molekül veri kümesini yükleyelim
>> dset = QM9('.')
>> len(dset) 130831
>> # Torch geometrik verileri şu şekilde sarar
>>> veri = dset[0]
>>> veri
Data(edge_attr=[8, 4], edge_index=[2, 8], idx=[1], name="gdb_1", pos=[5, 3],
x=[5, 11], y=[1, 19], z=[5])
>> # özniteliklere doğrudan erişebilir
>> data.z
tensör([6, 1, 1, 1, 1 ])
>> # her atomun atom numarası öznitelik ekleyebilir
>> data.new_attribute = torch.tensor([1, 2, 3])
>>> veri
Data(edge_attr=[8, 4], edge_index=[2, 8], idx=[1], name="gdb_1", new_
attribute=[3], pos=[5, 3], x=[5, 11], y=[1, 19], z=[5])
>> # tüm öznitelikleri cihazlar arasında taşıyabilir
>>> cihaz = torch.device(
...         "cuda:0" if torch.cuda.is_available() else "cpu"
... )
>> data.to(cihaz)
>> data.new_attribute.is_cuda
True
```

Data nesnesi, grafik verileri için kullanışlı ve esnek bir sarmalayıcıdır. Birçok PyTorch Geometrik nesnesinin doğru şekilde işlenebilmesi için veri nesnelerinde belirli anahtar kelimelere ihtiyaç duyduğunu unutmayın. Özellikle, x düğüm özellikleri içermeli, edge_attr kenar özellikleri içermeli, edge_index bir kenar listesi içermeli ve y etiketler içermelidir. QM9 verileri bazı ek özellikler içerir: pos, moleküllerin atomlarının her birinin 3B ızgaradaki konumu ve z, moleküldeki her atomun atom numarası. QM9'daki etiketler, dipol moment, serbest enerji, entalpi veya izotropik polarizasyon gibi moleküllerin bir grup fiziksel özelliğidir. Bir GNN uygulayacağız ve izotropik polarizasyonu tahmin etmek için QM9 üzerinde eğiteceğiz.

 alinpikbarbell@gmail.com

QM9 veri seti

QM9 veri kümesi, çeşitli geometrik, enerjik, elektronik ve termodinamik özelliklerle etiketlenmiş 133.885 küçük organik molekül içerir. QM9, kimyasal yapı-özellik ilişkilerini ve hibrit kuantum mekaniği/makine öğrenme yöntemlerini tahmin etmeye yönelik yöntemler geliştirmek için ortak bir kıyaslama veri kümesidir. Veri seti hakkında daha fazla bilgi http://quantum-machine.org/datasets/ adresinde bulunabilir.

Moleküllerin bağ türleri önemlidir; yani, hangi atomların belirli bir bağ türüyle, örneğin tek veya çift bağlarla bağlandığı önemlidir. Bu nedenle, kenar özelliklerini kullanabilen bir grafik konvolüsyonu kullanmak isteyeceğiz. Bunun için torch_geometric.nn.NNConv katmanını kullanacağız. (Uygulama ayrıntılarıyla ilgileniyorsanız, kaynak kodu https://pytorch-geometric.readthedocs.io/en/latest/_modules/torch_geometric/nn/conv/nn_conv.html#NNConv adresinde bulunabilir).

NNConv katmanındaki bu konvolüsyon aşağıdaki formu alır:

$$x_i^{(t)} = W_i x_i^{(t-1)} + \sum_{j \in N(i)} x_j^{(t-1)} \cdot h_\Theta(e_{i,j})$$

Burada, *h bir* dizi Θ ağırlığıyla parametrelendirilmiş bir sinir ağıdır ve W düğüm etiketleri için bir ağırlık matrisidir. Bu grafik konvolüsyonu, daha önce sıfırdan uyguladığımıza çok benzemektedir:

$$x_i^{(t)} = W_1 x_i^{(t-1)} + \sum_{j \in N(i)} x_j^{(t-1)} W_2$$

Tek gerçek fark, W_2 eşdeğeri olan sinir ağı *h'nin,* farklı kenar etiketleri için ağırlıkların değişmesine izin veren kenar etiketlerine göre parametrelendirilmiş olmasıdır. Aşağıdaki kod aracılığıyla, bu tür iki grafik evrişimli katman (NNConv) kullanan bir GNN uyguluyoruz:

```
sınıf ExampleNet(torch.nn.Module):
    def init (self, num_node_features, num_edge_features):
        super(). init ()
        conv1_net = nn.Sequential(
            nn.Linear(num_edge_features, 32),
            nn.ReLU(),
            nn.Linear(32, num_node_features*32))

        conv2_net = nn.Sequential(
            nn.Linear(num_edge_features, 32),
            nn.ReLU(),
            nn.Linear(32, 32*16))

        self.conv1 = NNConv(num_node_features, 32, conv1_net)
        self.conv2 = NNConv(32,16, conv2_net)
```

```
        self.fc_1 = nn.Linear(16, 32)
        self.out = nn.Linear(32, 1)

    def forward(self, data):
        batch, x, edge_index, edge_attr = (
            data.batch, d a t a . x, d a t a . edge_index, data.edge_attr)
        # İlk grafik conv katmanı
        x = F.relu(self.conv1(x, edge_index, edge_attr))
        # İkinci grafik conv katmanı
        x = F .relu(self.conv2(x, edge_index, edge_attr)) x =
        global_add_pool(x,batch)
        x = F.relu(self.fc_1(x))
        output = self.out(x)
        return output
```

Bu GNN'yi, bir molekülün yük dağılımının harici bir elektrik alanı tarafından bozulma eğiliminin bir ölçüsü olan molekülün izotropik polarize edilebilirliğini tahmin etmek için eğiteceğiz. QM9 veri kümesini eğitim, doğrulama ve test kümelerine ayıracağız ve PyTorch Geometric DataLoader'ı kullanacağız. Bunların özel bir collate fonksiyonu gerektirmediğini, ancak uygun şekilde adlandırılmış niteliklere sahip bir Data nesnesi gerektirdiğini unutmayın.

Ardından, veri kümesini bölelim:

```
>> from torch.utils.data import random_split
>> train_set, valid_set, test_set = random_split(
...         dset,[110000, 10831, 10000])
>> trainloader = DataLoader(train_set, batch_size=32, shuffle=True)
>> validloader = DataLoader(valid_set, batch_size=32, shuffle=True)
>> testloader = DataLoader(test_set, batch_size=32, shuffle=True)
```

Aşağıdaki kod bir ağı GPU üzerinde başlatacak ve eğitecektir (eğer mevcutsa):

```
>> # bir ağı başlatın
>> qm9_node_feats, qm9_edge_feats = 11, 4
>> net = ExampleNet(qm9_node_feats, qm9_edge_feats)

>> # bazı makul parametrelerle bir optimize edici başlatın
>>> optimizer = torch.optim.Adam(
...         net.parameters(), lr=0.01)
>>> epochs = 4
>> target_idx = 1 # polarize edilebilirlik etiketinin indeks konumu
>> cihaz = torch.device("cuda:0" if
...                                 torch.cuda.is_available() else "cpu")
>> net.to(cihaz)
```

 idinpikbarbell@gmail.com

Aşağıdaki kodda gösterilen eğitim döngüsü, önceki PyTorch bölümlerinde karşılaştığımız tanıdık modeli takip eder, bu nedenle açıklama ayrıntılarını atlayabiliriz. Bununla birlikte, vurgulanması gereken bir ayrıntı, polarize edilebilirlik bir sınıf etiketi değil sürekli bir hedef olduğundan, burada çapraz tropi yerine ortalama karesel hata (MSE) kaybını hesaplıyor olmamızdır:

```
>> for total_epochs in range(epochs):
...     epoch_loss = 0
...     total_graphs = 0
...     net.train()
...     tren yükleyicide parti için:
...         batch.to(cihaz)
...         optimizer.zero_grad()
...         çıktı = net(batch)
...         kayıp = F.mse_loss(
...             output,batch.y[:, target_idx].unsqueeze(1))
...         loss.backward()
...         epoch_loss += loss.item()
...         total_graphs += batch.num_graphs
...         optimizer.step()
...     train_avg_loss = epoch_loss / total_graphs
...     val_loss = 0
...     total_graphs = 0
...     net.eval()
...     for batch in validloader:
...         batch.to(cihaz)
...         çıktı = net(batch)
...         kayıp = F.mse_loss(
...             output,batch.y[:, target_idx].unsqueeze(1))
...         val_loss += loss.item()
...         total_graphs += batch.num_graphs
...     val_avg_loss = val_loss / total_graphs
...     print(f "Epochs: {total_epochs} | "
...           f "epoch avg. loss: {train_avg_loss:.2f} | "
...           f "doğrulama avg. kayıp: {val_avg_loss:.2f}")
Epochs: 0 | epoch ortalama kaybı: 0,30 | doğrulama
ortalama kaybı: 0,10 Epochs: 1 | epoch ortalama kaybı: 0,12
| doğrulama ortalama kaybı: 0,07 Dönemler: 2 | dönem
ortalama kaybı: 0,10 | doğrulama ortalama kaybı: 0,05
Dönemler: 3 | dönem ort. kaybı: 0,09 | doğrulama ortalama
kaybı: 0,07
```

İlk dört eğitim dönemi boyunca, hem eğitim hem de doğrulama kaybı azalmaktadır. Veri kümesi büyüktür ve bir CPU'da eğitilmesi biraz zaman alabilir, bu nedenle dört epoktan sonra eğitimi durdururuz. Ancak, modeli daha fazla eğitirsek, kayıp iyileşmeye devam edecektir. Bunun performansı nasıl değiştirdiğini görmek için modeli ek epoklar için eğitebilirsiniz.

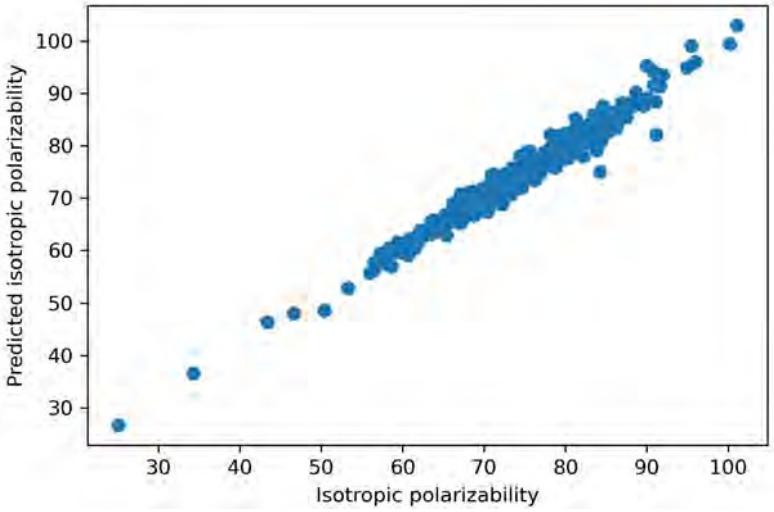
Aşağıdaki kod, test verileri üzerindeki değerleri tahmin eder ve gerçek etiketleri toplar:

```
>> net.eval()
>>> tahminler = []
>> real = []
>>> for batch in testloader:
...     çıktı = net(batch.to(device))
...     predictions.append(output.detach().cpu().numpy())
...     real.append(
...             batch.y[:,target_idx] .detach().cpu().numpy())
>> real = np.concatenate(real)
>>> tahminler = np.concatenate(tahminler)
```

Şimdi test verilerinin bir alt kümesiyle bir dağılım grafiği oluşturabiliriz. Test veri kümesi nispeten büyük olduğundan (10.000 molekül), sonuçlar biraz dağınık olabilir ve basitlik için yalnızca ilk 500 tahmin ve hedefi çiziyoruz:

```
>> import matplotlib.pyplot as plt
>> plt.scatter(real[:500], predictions[:500])
>> plt.xlabel('İzotropik polarize edilebilirlik')
>> plt.ylabel('Tahmin edilen izotropik polarize edilebilirlik')
```

Ortaya çıkan şekil burada gösterilmektedir:



Şekil 18.12: Tahmin edilen izotropik polarizabilite ile gerçek izotropik polarizabilitenin grafiği

 adinpikbarbell@gmail.com

Çizime dayanarak, noktaların nispeten köşegene yakın olduğu göz önüne alındığında, basit GNN'mizin hiperparametre ayarı olmadan bile izotropik polarizasyon değerlerini tahmin etmede iyi bir iş çıkardığı görülmektedir.

TorchDrug - İlaç keşfi için PyTorch tabanlı bir kütüphane

PyTorch Geometric, bu bölümde gördüğünüz gibi moleküller de dahil olmak üzere grafiklerle çalışmak için kapsamlı bir genel amaçlı kütüphanedir. Daha derinlemesine molekül çalışması ve ilaç keşfi ile ilgileniyorsanız, moleküllerle çalışmak için birçok kullanışlı yardımcı program sunan yakın zamanda geliştirilen TorchDrug kütüphanesini de göz önünde bulundurmanızı öneririz. TorchDrug hakkında daha fazla bilgiyi burada bulabilirsiniz: https://torchdrug.ai/.

Diğer GNN katmanları ve son gelişmeler

Bu bölümde GNN'lerinizde kullanabileceğiniz bazı ek katmanlar tanıtılacak ve bu alandaki son gelişmelere dair üst düzey bir genel bakış sunulacaktır. Bu katmanların ve uygulamalarının arkasındaki sezgiler hakkında arka plan sağlarken, bu kavramlar matematiksel olarak biraz karmaşık hale gelebilir, ancak cesaretiniz kırılmasın. Bunlar isteğe bağlı konulardır ve tüm bu uygulamaların inceliklerini kavramak gerekli değildir. Katmanların arkasındaki genel fikirleri anlamak, referans verdiğimiz PyTorch Geometrik uygulamalarını denemek için yeterli olacaktır.

Aşağıdaki alt bölümlerde spektral çizge konvolüsyon katmanları, çizge havuzlama katmanları ve çizgeler için normalizasyon katmanları tanıtılacaktır. Son olarak, son alt bölüm bazı daha gelişmiş çizge sinir ağı türlerine kuş bakışı bir bakış sağlayacaktır.

Spektral grafik konvolüsyonları

Bu noktaya kadar kullandığımız grafik konvolüsyonlarının hepsi doğası gereği uzaysaldı. Bu, grafikle ilişkili topolojik uzaya dayalı olarak bilgi topladıkları anlamına gelir; bu, uzamsal konvolüsyonların düğümlerin yerel komşulukları üzerinde çalıştığını söylemenin süslü bir yoludur. Bunun bir sonucu olarak, uzamsal konvolüsyonları kullanan bir GNN'nin grafik verilerindeki karmaşık küresel paternleri yakalaması gerekiyorsa, ağın birden fazla uzamsal konvolüsyonu yığması gerekecektir. Bu küresel örüntülerin önemli olduğu ancak ağ derinliğinin sınırlandırılması gereken durumlarda, spektral çizge konvolüsyonları dikkate alınması gereken alternatif bir konvolüsyon türüdür.

Spektral grafik konvolüsyonları uzaysal grafik konvolüsyonlarından farklı çalışır. Spektral grafik konvolüsyonları, grafiğin *Laplacian* adı verilen bitişiklik matrisinin normalleştirilmiş bir versiyonunun eigende- bileşimini hesaplayarak *grafiğin* spektrumunu (özdeğerler kümesi) kullanarak çalışır. Bu son cümle biraz zor gibi görünebilir, o yüzden biraz daha açalım ve adım adım üzerinden geçelim.

Yönlendirilmemiş bir çizge için, bir çizgenin Laplacian matrisi $\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{A}$ olarak tanımlanır; burada ***A çizgenin bitişiklik matrisi*** ve ***D derece*** matrisidir. Derece matrisi köşegen bir matristir ve *i* indeksli satırdaki köşegen üzerindeki eleman, bitişiklik matrisinin *i'inci satırıyla* ilişkili düğüme giren ve çıkan kenarların sayısıdır.

L reel değerli bir simetrik matristir ve reel değerli simetrik matrislerin L $\boldsymbol{L}$ $\boldsymbol{Q\Lambda Q}^T$ şeklinde ayrıştırılabileceği kanıtlanmıştır; burada ***Q***, sütunları ***L***'nin özvektörleri olan ortogonal bir matristir ve **Λ,** elemanları ***L***'nin özdeğerleri olan diyagonal bir matristir.

Grafiğin yapısının temel bir temsilini sağlar. ***A*** tarafından tanımlanan grafiğin yerel komşuluklarını kullanan uzamsal konvolüsyonların aksine, spektral konvolüsyonlar düğüm katıştırmalarını güncellemek için ***Q'***dan yapının alternatif temsilini kullanır.

Aşağıdaki spektral konvolüsyon örneği, bir grafik için aşağıdaki gibi tanımlanan *simetrik normalleştirilmiş grafik Laplacian'ın* eigendecomposition'ını kullanır:

$$_{Lsym} = I \boldsymbol{I} \; D^{-\frac{1}{2}} AD^{-\frac{1}{2}}$$

Burada, ***I*** kimlik matrisidir. Bu, grafik Laplacian'ın normalleştirilmesi, özellik standardizasyonuna benzer şekilde gradyan tabanlı eğitim prosedürünün stabilize edilmesine yardımcı olabileceği için kullanılır.

^{T} $\boldsymbol{Q\Lambda Q}_{sym}$ 'nin eigendecomposition'ı olduğu göz önüne alındığında, grafik konvolüsyonu aşağıdaki gibi tanımlanır:

$$'XX = Q(Q^T XXXQ^T WW)$$

Burada, ***W*** eğitilebilir bir ağırlık matrisidir. Parantezlerin içi esasen ***X*** ve ***W'***yi grafikteki yapısal ilişkileri kodlayan bir matrisle çarpmaktadır. Buradaki ⊙ operatörü iç terimlerin eleman bazında çarpımını gösterirken, dıştaki ***Q*** sonucu orijinal temele geri eşler. Bu konvolüsyonun birkaç istenmeyen özelliği vardır, çünkü bir grafın eigendecomposition'ını hesaplamak $O(n^3)$ hesaplama karmaşıklığına sahiptir. Bu, yavaş olduğu ve yapılandırılmış olduğu için ***W'nin*** grafın boyutuna bağlı olduğu anlamına gelir. Sonuç olarak, spektral konvolüsyon yalnızca aynı boyuttaki graflara uygulanabilir. Ayrıca, bu konvolüsyonun alıcı alanı tüm graftır ve bu mevcut formülasyonda ayarlanamaz. Bununla birlikte, bu sorunları ele almak için çeşitli teknikler ve konvolüsyonlar geliştirilmiştir.

Örneğin, Bruna ve meslektaşları (https://arxiv.org/abs/1312.6203), ***W***'nin boyut bağımlılığını, her biri kendi skaler parametresi olan α ile çarpılan bir dizi fonksiyonla yaklaştırarak ele alan bir yumuşatma yöntemi sunmuştur. Yani, $f_1, \ldots, f_n$, $\boldsymbol{WW}\,\boldsymbol{W} \sum \alpha_i f_i$ fonksiyonları kümesi verildiğinde. Fonksiyon kümesi, boyutluluğun değiştirilebileceği şekildedir. Bununla birlikte, α skaler kaldığından, konvolüsyon parametre uzayı grafik boyutundan bağımsız olabilir.

Bahsetmeye değer diğer spektral konvolüsyonlar arasında Chebyshev grafik konvolüsyonu (https:// arxiv.org/abs/1606.09375) bulunmaktadır; bu konvolüsyon orijinal spektral konvolüsyona daha düşük bir zaman karmaşıklığında yaklaşabilir ve farklı boyutlarda alıcı alanlara sahip olabilir. Kipf ve Welling (https://arxiv. org/abs/1609.02907) Chebyshev konvolüsyonlarına benzer özelliklere sahip, ancak daha az parametre yükü olan bir konvolüsyon sunmaktadır. Bunların her ikisinin de uygulamaları PyTorch Geometric'te torch_geometric.nn.ChebConv ve torch_geometric.nn.GCNConv olarak mevcuttur ve spektral konvolüsyonlarla oynamak istiyorsanız başlamak için makul yerlerdir.

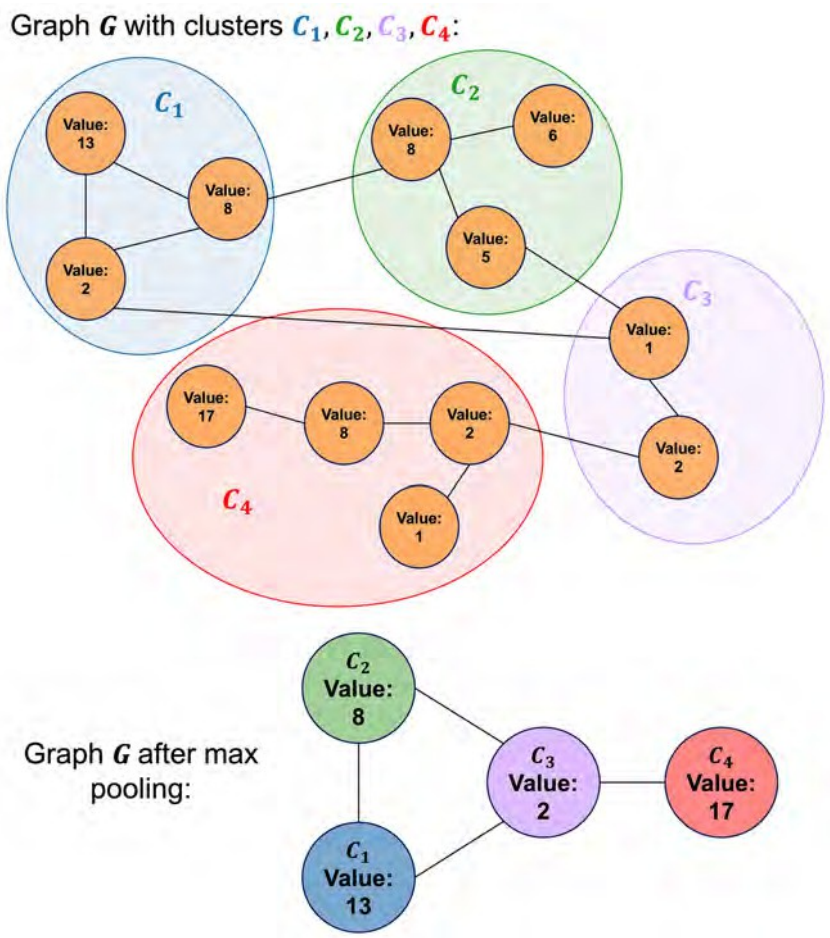
 alinpikbarbell@gmail.com

Havuzlama

Grafikler için geliştirilmiş bazı havuzlama katmanı örneklerini kısaca tartışacağız. Havuzlama katmanları tarafından sağlanan alt örnekleme CNN mimarilerinde faydalı olurken, GNN'lerde alt örneklemenin faydası bu kadar net bir şekilde fark edilmemiştir.

Görüntü verileri için havuzlama katmanları, grafiklerin sahip olmadığı uzamsal yerelliği (ab)kullanır. Bir grafikteki düğümlerin kümelenmesi sağlanırsa, bir grafik havuzlama katmanının düğümleri nasıl havuzlaması gerektiğini tanımlayabiliriz. Bununla birlikte, optimum kümelemenin nasıl tanımlanacağı belirsizdir ve farklı bağlamlar için farklı kümeleme yaklaşımları tercih edilebilir. Kümeleme belirlendikten sonra bile, düğümler aşağı örneklenirse, kalan düğümlerin nasıl bağlanması gerektiği belirsizdir. Bunlar hala açık araştırma soruları olsa da, birkaç çizge havuzlama katmanına bakacağız ve yukarıda belirtilen sorunlara yaklaşımlarına işaret edeceğiz.

CNN'lerde olduğu gibi, GNN'lere uygulanabilecek ortalama ve maksimum havuzlama katmanları vardır. *Şekil 18.13*'te gösterildiği gibi, bir düğüm kümesi verildiğinde, her küme yeni bir grafikte bir düğüm haline gelir:



Şekil 18.13: Bir grafiğe maksimum havuzlama uygulanması

Her kümenin katıştırması, kümedeki düğümlerin katıştırmalarının ortalamasına veya maksimumuna eşittir. Bağlanabilirliği ele almak için kümeye, kümedeki tüm kenar indekslerinin birleşimi atanır. Örneğin, *i, j*, *k düğümleri* c_1 kümesine atanmışsa, *i*, *j* veya *k ile bir kenar* paylaşan herhangi bir düğüm veya düğüm içeren *küme* c_1 ile bir kenar paylaşacaktır.

Daha karmaşık bir havuzlama katmanı olan *DiffPool* (https://arxiv.org/abs/1806.08804), hem kümeleme hem de alt örneklemeyi aynı anda ele almaya çalışır. Bu katman yumuşak bir küme atama matrisi öğrenir

S S^{Snnc}, *n* düğüm katıştırmasını *c* kümeye dağıtır. (Yumuşak ve sert hakkında bir tazeleme için kümeleme, *Bölüm 10*, *Etiketlenmemiş Verilerle Çalışma - Kümeleme Analizi*'ndeki *Sert ve soß kümeleme* bölümüne bakın). Bununla $X, X' = S^T X$ ve A, $A' = S A^{TT}$ S olarak güncellenir. Özellikle, A' artık ayrık değerler içermez ve bunun yerine kenar ağırlıklarının bir matrisi olarak görülebilir. Zaman içinde *DiffPool,* yorumlanabilir yapıya sahip neredeyse zor bir kümeleme atamasına yakınsar.

Bir başka havuzlama yöntemi olan *top-k* havuzlama, düğümleri toplamak yerine grafikten düşürerek kümelenme ve bağlantı sorunlarını ortadan kaldırır. Bu görünüşte atılan düğümlerdeki bilginin kaybıyla birlikte gelse de, bir ağ bağlamında, havuzlamadan önce bir konvolüsyon gerçekleştiği sürece, ağ bundan kaçınmayı öğrenebilir. Düşen düğümler, öğrenilebilir bir *p* vektörüne karşı bir projeksiyon puanı kullanılarak seçilir. *Towards Sparse Hierarchical Graph Classifiers*'da (https://arxiv.org/abs/1811.01287) belirtildiği gibi (X', A') hesaplamak için gerçek formülasyon şöyledir:

$$y \; y \; \frac{Xp}{\|p\|}, \; i \; y \; \text{top-k}(y, k), \; XX' \; y \; (XX \; \mathbf{tanh}(y)), \; A' \; y \; A$$

Burada, *top-k* y'nin indekslerini seçer, en iyi *k* değerleri ve indeks vektörü *i,* X ve A'nın satırlarını bırakmak için kullanılır. *Top-k* havuzlama PyTorch Geometric'te torch_geometric.nn.TopKPooling olarak uygulanır. Ek olarak, maksimum ve ortalama havuzlama sırasıyla torch_geometric.nn.max_pool_x ve torch_geometric.nn.avg_pool_x olarak uygulanır.

Normalleştirme

Normalizasyon teknikleri, eğitim sürecini stabilize etmeye ve/veya hızlandırmaya yardımcı olmak için birçok sinir ağı türünde kullanılmaktadır. Toplu normalizasyon gibi birçok yaklaşım (*Bölüm 17*, *Yeni Verilerin Sentezlenmesi için Üretken Çekişmeli Ağlar*'da tartışılmıştır), uygun defter tutma ile GNN'lerde kolayca uygulanabilir. Bu bölümde, özellikle grafik verileri için tasarlanmış bazı normalleştirme katmanlarını kısaca açıklayacağız.

Normalleştirmenin hızlı bir incelemesi olarak, bir dizi özellik değeri x_1 , ..., x_n verildiğinde, değerleri x_i-μ ile güncelleriz; burada μ değer kümesinin ortalaması ve σ standart sapmasıdır. Tipik olarak, çoğu sinir ağı normalleştirme yöntemleri genel formunu alır $\frac{x_i-\mu}{\sigma} + \beta$, burada γ ve β öğrenilebilirdir parametreleriyle ilgilidir ve yöntemler arasındaki fark, normalleştirmenin uygulandığı özellik kümesiyle ilgilidir.

GraphNorm: A Principled Approach to Accelerating Graph Neural Network Training by *Tianle Cai* and colleagues, 2020 (https://arxiv.org/abs/2009.03294), bir grafik konvolüsyonunda toplamadan sonra ortalama istatistiğin anlamlı bilgiler içerebileceğini, bu nedenle tamamen atılmasının arzu

adinpikbarbell@gmail.com

edilmeyebileceğini göstermiştir. Bunu ele almak için *GraphNorm'u* tanıttılar.

Orijinal el yazmasından ödünç alınan notasyonda, h düğüm gömme matrisi olsun. $h_{i,j}$ v_i düğümünün j. özellik değeri olsun, burada $i = 1, ..., n$ ve $j = 1, ..., d$. *GraphNorm* aşağıdaki formu alır:

$$\gamma_{jj} \frac{h_{ijj} - \alpha_{jj} \cdot \mu_{jj}}{\sigma_{jj}} + \beta_{jj}$$

Burada, $\mu_{jj} = \frac{\sum_{i=1}^{n} h_{iji}}{n}$ ve $\sigma_{jj}^2 = \frac{\sum_{i=1}^{n} (h_{iji} - \alpha_{jj}\mu_{jj})^2}{n}$. Anahtar ekleme, öğrenilebilir parametre olan α'dir. ortalama istatistiğin, μ_{jj} , ne kadarının atılacağını kontrol eder.

Bir başka çizge normalleştirme tekniği de *Guohao Li* ve arkadaşları tarafından *DeeperGCN: All You Need to Train Deeper GCNs* in 2020 (https://arxiv.org/ abs/2006.07739) adlı makalede açıklanan *MsgNorm'dur. MsgNorm, bu* bölümde daha önce bahsedilen grafik konvolüsyonlarının mesaj geçiren formülasyonuna karşılık gelmektedir. Mesaj geçiren ağ terminolojisini kullanarak (*temel bir grafik konvolüsyonunun uygulanması* alt bölümünün sonunda tanımlanmıştır), bir grafik konvolüsyonu M_t üzerinde toplandıktan ve m_i ürettikten sonra, ancak U_t ile gömülen düğümleri güncellemeden önce, *MsgNorm* m_i 'yi aşağıdaki formülle normalleştirir:

$$m'_i = s \cdot s \|h_i\|_2 \cdot \frac{m_i}{\|m_i\|_2}$$

Burada, s öğrenilebilir bir ölçeklendirme faktörüdür ve bu yaklaşımın arkasındaki sezgi, bir grafik konvolüsyonunda toplanmış mesajların özelliklerini normalleştirmektir. Bu normalleştirme yaklaşımını destekleyen bir teori olmasa da, pratikte iyi sonuç vermiştir.

Tartıştığımız normalleştirme katmanlarının tümü PyTorch Geomet- ric aracılığıyla BatchNorm, GroupNorm ve MessageNorm olarak uygulanır ve kullanılabilir. Daha fazla bilgi için lütfen https://pytorch-geometric.readthedocs.io/en/latest/modules/ nn.html#normalization-layers adresindeki PyTorch Geometric belgelerini ziyaret edin.

Ek bir kümeleme kurulumu gerektirebilen grafik havuzlama katmanlarının aksine, grafik normalleştirme katmanları mevcut bir GNN modeline daha kolay takılabilir. Model geliştirme ve optimizasyon sırasında çeşitli normalleştirme yöntemlerinin test edilmesi makul ve önerilen bir yaklaşımdır.

Gelişmiş çizge sinir ağı literatürüne işaretçiler

Graflara odaklanan derin öğrenme alanı hızla gelişmektedir ve bu giriş bölümünde makul ayrıntılarla ele alamayacağımız birçok yöntem vardır. Bu nedenle, bu bölümü sonlandırmadan önce, ilgilenen okuyuculara bu konuyla ilgili daha derinlemesine çalışmalar için kayda değer literatüre bir dizi işaretçi sunmak istiyoruz.

Bölüm 16, *Dönüştürücüler - Doğal Dil İşlemenin Dikkat Mekanizmalarıyla Geliştirilmesi'n*den hatırlayacağınız gibi, dikkat mekanizmaları ek bağlamlar sağlayarak modellerin yeteneklerini geliştirebilir. Bu bağlamda, GNN'ler için çeşitli dikkat yöntemleri geliştirilmiştir. Dikkat ile güçlendirilmiş GNN'lere örnek olarak *Petar Veličković ve* meslektaşlarının 2017 tarihli *Graph Attention Networks (Grafik Dikkat Ağları*) (https://arxiv.org/abs/1710.10903) ve *Dan Busbridge ve* meslektaşlarının 2019 tarihli *Relational Graph Attention Networks* (*İlişkisel Grafik Dikkat Ağları*) (https://arxiv.org/abs/1904.05811) verilebilir.

 alinpikbarbell@gmail.com

Son zamanlarda, bu dikkat mekanizmaları *Seongjun Yun ve* meslektaşları tarafından önerilen çizge dönüştürücülerinde, 2020 (https://arxiv.org/abs/1911.06455) ve *Ziniu Hu ve* meslektaşları tarafından önerilen *Heterojen Çizge Dönüştürücüsünde,* 2020 (https://arxiv.org/abs/2003.01332) de kullanılmıştır.

Yukarıda bahsedilen çizge dönüştürücülerin yanı sıra, özellikle çizgeler için başka derin üretken modeller de geliştirilmiştir. *Kipf* ve *Welling*, 2016 (https://arxiv.org/abs/1611.07308) tarafından *Variation- al Graph Auto-Encoders,* Qi Liu ve meslektaşları, 2018 (https://arxiv.org/ abs/1805.09076) tarafından *Constrained Graph Variational Autoencoders for Molecule Design* ve *Simonovsky ve Komodakis*, 2018 (https://arxiv.org/abs/1802.03480) tarafından *GraphVAE: Towards Generation of Small Graphs Using Variational Autoencoders'*da tanıtılanlar gibi grafik *varyasyonel otomatik* kodlayıcılar vardır. *Molekül* üretimine uygulanan bir diğer önemli çizge varyasyonel otomatik kodlayıcı, *Wengong Jin* ve meslektaşları tarafından hazırlanan *Moleküler Çizge Üretimi için Bağlantı Ağacı Varyasyonel Otomatik Kodlayıcıdır,* 2019 (https://arxiv.org/ abs/1802.04364).

Bazı GAN'lar grafik verileri üretmek için tasarlanmıştır, ancak bu yazı itibariyle, GAN'ların grafikler üzerindeki performansı görüntü alanından çok daha az ikna edicidir. Örnekler arasında *Hongwei Wang* ve meslektaşlarının *GraphGAN: Graph Representation Learning with Generative Adversarial Nets,* 2017 (https://arxiv.org/abs/1711.08267) ve *Cao* ve *Kipf'in MolGAN: An Implicit Generative Model for Small Molecular Graphs,* 2018 (https://arxiv.org/abs/1805.11973) yer almaktadır.

GNN'ler ayrıca derin takviyeli öğrenme modellerine de dahil edilmiştir - bir sonraki bölümde takviyeli öğrenme hakkında daha fazla bilgi edineceksiniz. Örnekler arasında *Jiaxuan You* ve meslektaşları tarafından *Hedefe Yönelik Moleküler Grafik Üretimi* için Grafik Evrişimli *Politika Ağı,* 2018 (https://arxiv.org/ abs/1806.02473) ve molekül oluşturma görevlerine uygulanan bir GNN kullanan *Zhenpeng Zhou* ve meslektaşları tarafından *Derin Takviye Öğrenimi yoluyla* Moleküllerin *Optimizasyonunda* önerilen derin bir Q-ağ, 2018 (https://arxiv.org/abs/1810.08678) yer almaktadır.

Son olarak, teknik olarak grafik verisi olmasa da, 3D nokta bulutları bazen kenarları oluşturmak için mesafe kesmeleri kullanılarak bu şekilde temsil edilir. Çizge ağlarının bu alandaki uygulamaları arasında *Point-GNN bulunmaktadır:* LiDAR nokta bulutlarındaki 3B nesneleri tespit eden *Weijing Shi* ve meslektaşlarının 2020 tarihli *Graph Neural Network for 3D Object Detection in a Point Cloud çalışması* (https:// arxiv.org/abs/2003.01251). Ayrıca, *GAPNet: Graph Attention based Point Neural Network for Exploiting Local Feature of Point Cloud* by *Can Chen* and colleagues, 2019 (https://arxiv.org/abs/1905.08705), diğer derin mimariler için zor olan nokta bulutu verilerindeki yerel özellikleri tespit etmek için tasarlanmıştır.

6587191CA584E

Özet

Erişebildiğimiz veri miktarı artmaya devam ettikçe, veri içindeki ilişkileri anlama ihtiyacımız da artacaktır. Bu çeşitli şekillerde yapılsa da, grafikler bu ilişkilerin damıtılmış bir temsili olarak işlev görür, bu nedenle mevcut grafik verilerinin miktarı yalnızca artacaktır.

Bu bölümde, sıfırdan bir çizge konvolüsyon katmanı ve bir GNN uygulayarak çizge sinir ağlarını temelden açıkladık. Çizge verilerinin doğası gereği GNN'lerin uygulanmasının aslında oldukça karmaşık olduğunu gördük. Bu nedenle, GNN'leri moleküler polarizasyonu tahmin etmek gibi gerçek dünyadan bir örneğe uygulamak için, ihtiyacımız olan yapı taşlarının çoğunun uygulamalarını sağlayan PyTorch Geometric kütüphanesini nasıl kullanacağımızı öğrendik. Son olarak, GNN literatürüne daha derinlemesine dalmak için bazı önemli literatürü gözden geçirdik.

Umarım bu bölüm, derin öğrenmenin grafikler üzerinde öğrenmek için nasıl kullanılabileceğine dair bir giriş sağlamıştır. Bu alandaki yöntemler şu anda sıcak bir araştırma alanıdır ve bahsettiklerimizin çoğu son birkaç yıl içinde yayınlanmıştır. Bir başlangıç noktası olarak bu metinle, belki de bu alandaki bir sonraki ilerleme sizin tarafınızdan yapılabilir.

Bir sonraki bölümde, bu kitapta şimdiye kadar ele aldıklarımıza kıyasla tamamen farklı bir makine öğrenimi kategorisi olan pekiştirmeli öğrenmeye bakacağız.

Kitabımızın Discord alanına katılın

Benzer düşünen insanlarla tanışmak ve 2000'den fazla üyemizle birlikte öğrenmek için Discord topluluğumuza katılın:
https://packt.link/MLwPyTorch

6587191CA584E
adimpikbarbell@gmail.com

olimpikbarbell@gmail.com

19

Karmaşık Ortamlarda Karar Verme için Takviyeli Öğrenme

Önceki bölümlerde denetimli ve denetimsiz makine öğrenimine odaklandık. Ayrıca, bu tür makine öğrenimi ile karşılaşılan sorunların üstesinden gelmek için yapay sinir ağlarından ve derin öğrenmeden nasıl yararlanacağımızı da öğrendik. Hatırlayacağınız gibi, denetimli öğrenme, belirli bir girdi özellik vektöründen bir kategori etiketini veya sürekli değeri tahmin etmeye odaklanır. Denetimsiz öğrenme ise verilerden örüntüler çıkarmaya odaklanarak veri sıkıştırma (*Bölüm 5*, *Boyut Azaltma Yoluyla Verileri Sıkıştırma),* kümeleme (Bölüm *10*, *Etiketsiz Verilerle Çalışma - Kümeleme Analizi)* veya yeni veriler oluşturmak için eğitim seti dağılımına yaklaşma (*Bölüm 17*, *Yeni Verileri Sentezlemek için Üretken Çekişmeli Ağlar*) için kullanışlı hale gelir.

Bu bölümde, dikkatimizi makine öğreniminin ayrı bir kategorisi olan ve genel bir ödülü optimize etmek için *bir dizi eylemi* öğrenmeye odaklandığı için önceki kategorilerden farklı olan **takviyeli öğrenmeye** (**RL**) çeviriyoruz - örneğin, bir satranç oyununda kazanmak. Özet olarak, bu bölüm aşağıdaki konuları kapsayacaktır:

- Karmaşık ortamlarda karar vermeye yardımcı olmak için RL'nin temellerini öğrenmek, aracı/çevre etkileşimlerine aşina olmak ve ödül sürecinin nasıl işlediğini anlamak
- RL problemlerinin farklı kategorilerinin, model tabanlı ve modelsiz öğrenme görevlerinin, Monte Carlo ve zamansal fark öğrenme algoritmalarının tanıtılması
- Bir Q-öğrenme algoritmasının tablo formatında uygulanması
- RL problemlerini çözmek için fonksiyon yaklaşımını anlamak ve *derin* bir Q-öğrenme algoritması uygulayarak RL'yi derin öğrenme ile birleştirmek

RL karmaşık ve geniş bir araştırma alanıdır ve bu bölüm temel konulara odaklanmaktadır. Bu bölüm bir giriş niteliği taşıdığından ve dikkatimizi önemli yöntemler ve algoritmalar üzerinde tutmak için, temel kavramları gösteren temel örneklerle çalışacağız. Bununla birlikte, bu bölümün sonuna doğru, daha zorlu bir örneğin üzerinden geçeceğiz ve derin Q-öğrenme olarak bilinen belirli bir RL yaklaşımı için derin öğrenme mimarilerini kullanacağız.

Giriş - deneyimlerden öğrenme

Bu bölümde, öncelikle makine öğreniminin bir dalı olarak RL kavramını tanıtacağız ve makine öğreniminin diğer görevleriyle karşılaştırıldığında temel farklılıklarını göreceğiz. Daha sonra, bir RL sisteminin işlevsel bileşenlerini ele alacağız. Ardından, Markov karar sürecine dayalı RL matematiksel formülasyonunu göreceğiz.

Pekiştirmeli öğrenmeyi anlama

Bu noktaya kadar, bu kitap öncelikle *denetimli* ve *denetimsiz* öğrenmeye odaklanmıştır. *Denetimli* öğrenmede, bir gözetmen veya insan uzman tarafından sağlanan etiketli eğitim örneklerine dayandığımızı ve amacın görünmeyen, etiketsiz test örneklerine iyi genelleme yapabilen bir model eğitmek olduğunu hatırlayın. Bu, denetimli öğrenme modelinin belirli bir girdi örneğine denetçi insan uzmanla aynı etiketleri veya değerleri atamayı öğrenmesi gerektiği anlamına gelir. Öte yandan, *denetimsiz öğrenmede* amaç, kümeleme ve boyut azaltma yöntemlerinde olduğu gibi bir veri kümesinin altında yatan yapıyı öğrenmek veya yakalamaktır; veya benzer bir temel dağılıma sahip yeni, sentetik eğitim örneklerinin nasıl oluşturulacağını öğrenmektir. RL, denetimli ve denetimsiz öğrenmeden önemli ölçüde farklıdır ve bu nedenle RL genellikle "makine öğreniminin üçüncü kategorisi" olarak kabul edilir.

RL'yi denetimli ve denetimsiz öğrenme gibi makine öğreniminin diğer alt görevlerinden ayıran temel unsur, RL'nin *etkileşim yoluyla öğrenme* kavramı etrafında merkezlenmiş olmasıdır. Bu, RL'de modelin bir *ödül fonksiyonunu* maksimize etmek için bir çevre ile etkileşimlerden öğrendiği anlamına gelir.

Bir ödül fonksiyonunu maksimize etmek su- pervised learning'deki kayıp fonksiyonunu minimize etme kavramıyla ilişkili olsa da, bir dizi eylemi öğrenmek için *doğru* etiketler RL'de önceden bilinmez veya tanımlanmaz - bunun yerine, bir oyunda kazanmak gibi belirli bir istenen sonuca ulaşmak için çevre ile etkileşimler yoluyla öğrenilmeleri gerekir. RL ile model (**ajan** olarak da adlandırılır) çevresiyle etkileşime girer ve bunu yaparak birlikte bir **bölüm** olarak adlandırılan bir dizi etkileşim oluşturur. Bu etkileşimler sayesinde ajan, çevre tarafından belirlenen bir dizi ödül toplar. Bu ödüller olumlu ya da olumsuz olabilir ve bazen bir bölümün sonuna kadar ajana açıklanmazlar.

Örneğin, bir bilgisayara satranç oynamayı ve insan oyunculara karşı kazanmayı öğretmek istediğimizi düşünün. Bilgisayar tarafından yapılan her bir satranç hamlesinin etiketleri (ödülleri) oyunun sonuna kadar bilinmez, çünkü oyun sırasında belirli bir hamlenin o oyunu kazanmakla mı yoksa kaybetmekle mi sonuçlanacağını bilmeyiz. Sadece oyunun sonunda geri bildirim belirlenir. Bu geri bildirim, eğer bilgisayar oyunu kazanırsa muhtemelen olumlu bir ödül olacaktır, çünkü temsilci genel olarak istenen sonuca ulaşmıştır; ve tam tersi, eğer bilgisayar oyunu kaybederse muhtemelen olumsuz bir ödül verilecektir.

 olimpikbarbell@gmail.com

Ayrıca, satranç oynama örneği göz önüne alındığında, girdi mevcut konfigürasyondur, örneğin satranç taşlarının tahta üzerindeki dizilimi. Çok sayıda olası girdi (sistemin durumları) göz önüne alındığında, her bir konfigürasyonu veya durumu pozitif veya negatif olarak etiketlemek imkansızdır. Bu nedenle, bir öğrenme sürecini tanımlamak için, her oyunun sonunda, istenen sonuca ulaşıp ulaşmadığımızı (oyunu kazanıp kazanmadığımızı) bildiğimizde ödüller (ya da cezalar) veririz.

Bu, RL'nin özüdür. RL'de bir ajana, bilgisayara ya da robota bir şeyleri *nasıl* yapacağını öğretemeyiz ya da öğretmeyiz; sadece ajanın *neyi* başarmasını istediğimizi belirleyebiliriz. Daha sonra, belirli bir denemenin sonucuna dayanarak, ajanın başarısına veya başarısızlığına bağlı olarak ödülleri belirleyebiliriz. Bu da RL'yi karmaşık ortamlarda, özellikle de problem çözme görevi bilinmeyen, açıklanması ya da tanımlanması zor bir dizi adım gerektirdiğinde karar verme açısından çok cazip hale getirir.

Oyunlardaki ve robotikteki uygulamaların yanı sıra, RL örnekleri doğada da bulunabilir. Örneğin, bir köpeği eğitmek RL'yi içerir - belirli arzu edilen eylemleri gerçekleştirdiğinde köpeğe ödüller (ikramlar) dağıtırız. Ya da partnerini yaklaşan bir nöbete karşı uyarmak üzere eğitilmiş bir sağlık köpeğini düşünün. Bu durumda, köpeğin yaklaşmakta olan bir nöbeti tespit edebildiği mekanizmayı tam olarak bilmiyoruz ve bu mekanizma hakkında kesin bilgiye sahip olsaydık bile, nöbet tespitini öğrenmek için bir dizi adım tanımlayamazdık. Bununla birlikte, bu davranışı *pekiştirmek* için bir nöbeti başarılı bir şekilde tespit ederse köpeği bir ödül mamasıyla ödüllendirebiliriz!

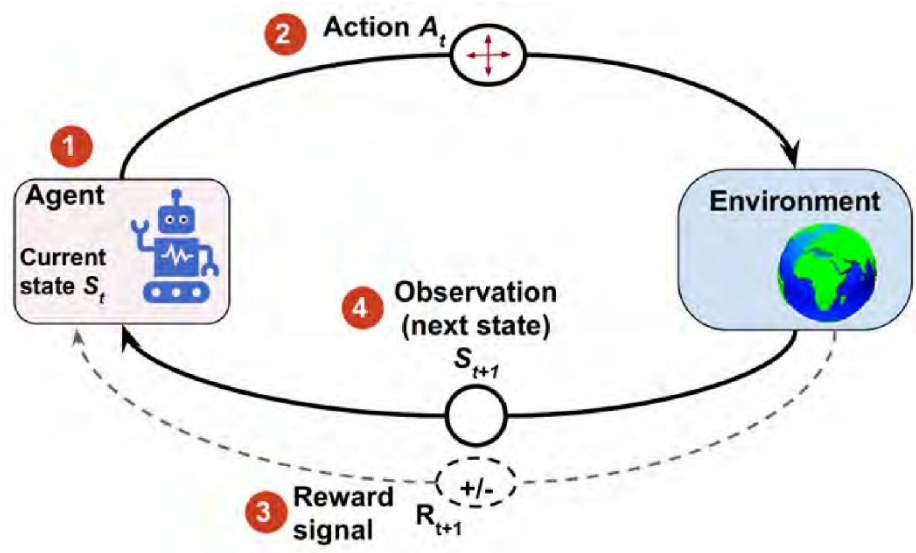
RL, belirli bir hedefe ulaşmak için rastgele bir dizi eylemi öğrenmek için güçlü bir çerçeve sağlarken, RL'nin hala çözülmemiş birçok zorluğu olan nispeten genç ve aktif bir araştırma alanı olduğunu lütfen unutmayın. RL modellerinin eğitimini özellikle zorlaştıran bir husus, sonuçta ortaya çıkan model girdilerinin daha önce gerçekleştirilen eylemlere bağlı olmasıdır. Bu, her türlü soruna yol açabilir ve genellikle dengesiz öğrenme davranışıyla sonuçlanır. Ayrıca, RL'deki bu sıra bağımlılığı, *gecikme etkisi olarak* adlandırılan bir *etki* yaratır; bu, bir *t* zaman adımında gerçekleştirilen eylemin, gelecekteki bir ödülün keyfi sayıda adım sonra ortaya çıkmasına neden olabileceği anlamına gelir.

Bir pekiştirmeli öğrenme sisteminin ajan-çevre arayüzünün tanımlanması

Tüm RL örneklerinde iki farklı varlık bulabiliriz: bir etmen ve bir çevre. Resmi olarak bir **etmen,** nasıl karar vereceğini öğrenen ve bir eylemde bulunarak çevresiyle etkileşime giren bir varlık olarak tanımlanır. Bunun karşılığında, eyleme geçmenin bir sonucu olarak, etmen gözlemler ve çevre tarafından yönetilen bir ödül sinyali alır. **Çevre**, etkenin dışında kalan her şeydir. Çevre, ajanla iletişim kurar ve ajanın eyleminin yanı sıra gözlemleri için ödül sinyalini belirler.

Ödül sinyali, ajanın çevreyle etkileşiminden aldığı geri bildirimdir ve genellikle skaler bir değer şeklinde sağlanır ve pozitif ya da negatif olabilir. Ödülün amacı, ajana ne kadar iyi performans gösterdiğini söylemektir. Temsilcinin ödülü alma sıklığı verilen görev ya da probleme bağlıdır. Örneğin, satranç oyununda ödül, tüm hamlelerin sonucuna göre tam bir oyundan sonra belirlenir: bir galibiyet veya bir mağlubiyet. Öte yandan, ödülün her zaman adımından sonra belirleneceği bir labirent tanımlayabiliriz. Böyle bir labirentte ajan, yaşam süresi boyunca birikmiş ödüllerini maksimize etmeye çalışır - burada yaşam süresi bir bölümün süresini tanımlar.

Şekil 19.1, aracı ile çevre arasındaki etkileşimleri ve iletişimi göstermektedir:



Şekil 19.1: Temsilci ve çevresi arasındaki etkileşim

Şekil 19.1'de gösterildiği gibi aracının durumu, tüm değişkenlerinin kümesidir (1). Örneğin, bir robot drone söz konusu olduğunda, bu değişkenler drone'un mevcut konumunu (enlem, boylam ve irtifa), drone'un kalan pil ömrünü, her bir fanın hızını ve benzerlerini içerebilir. Her zaman adımında, aracı bir dizi mevcut eylem $\boldsymbol{A_t}$ (2) aracılığıyla çevre ile etkileşime girer. $\boldsymbol{A_t}$ ile gösterilen ajan tarafından gerçekleştirilen eyleme bağlı olarak, ajan $\boldsymbol{S_t}$ durumundayken, ajan bir ödül sinyali $\boldsymbol{R_{t+1}}$ (3) alacak ve durumu $\boldsymbol{S_{t+1}}$ (4) haline gelecektir.

Öğrenme süreci boyunca, ajan farklı eylemleri denemelidir (**keşif),** böylece toplam, kümülatif ödülü en üst düzeye çıkarmak için hangi eylemleri tercih edeceğini ve daha sık gerçekleştireceğini (**istismar**) aşamalı olarak öğrenebilir. Bu kavramı anlamak için, yazılım mühendisliğine odaklanan yeni bir bilgisayar bilimi mezununun bir şirkette çalışmaya mı başlayacağını (sömürü) yoksa veri bilimi ve makine öğrenimi hakkında daha fazla bilgi edinmek için yüksek lisans veya doktora derecesi mi alacağını (keşif) merak ettiği çok basit bir örneği ele alalım. Genel olarak, sömürü kısa vadede daha fazla ödül getirecek eylemlerin seçilmesiyle sonuçlanırken, keşif uzun vadede potansiyel olarak daha fazla toplam ödül getirebilir. Keşif ve sömürü arasındaki denge kapsamlı bir şekilde incelenmiştir, ancak yine de bu karar verme ikilemine evrensel bir cevap yoktur.

RL'nin teorik temelleri

Bazı pratik örneklere geçmeden ve bu bölümün ilerleyen kısımlarında yapacağımız gibi bir RL modelini eğitmeye başlamadan önce, öncelikle RL'nin bazı teorik temellerini anlayalım. Aşağıdaki bölümler ilk olarak **Markov karar süreçlerinin** matematiksel formülasyonunu, epizodik ve sürekli görevleri, bazı temel RL terminolojisini ve **Bellman denklemini** kullanarak dinamik programlamayı inceleyerek başlayacaktır. Markov karar süreçleri ile başlayalım.

 olimpikbarbell@gmail.com

Markov karar süreçleri

Genel olarak, RL'nin ele aldığı problem türleri tipik olarak **Markov karar süreçleri** (**MDP'ler**) olarak formüle edilir. MDP problemlerini çözmek için standart yaklaşım dinamik programlama kullanmaktır, ancak RL dinamik programlamaya göre bazı önemli avantajlar sunar.

Dinamik programlama

Dinamik programlama, 1950'lerde Richard Bellman tarafından geliştirilen bir dizi bilgisayar algoritması ve programlama yöntemini ifade eder. Dinamik programlama bir anlamda özyinelemeli problem çözme ile ilgilidir - nispeten karmaşık problemleri daha küçük alt problemlere ayırarak çözmek.

Özyineleme ve dinamik programlama arasındaki temel fark, dinamik programlamanın alt problemlerin sonuçlarını (genellikle bir sözlük veya başka bir arama tablosu olarak) saklamasıdır, böylece gelecekte tekrar karşılaşıldığında bunlara sabit zamanda (yeniden hesaplamak yerine) erişilebilir.

Bilgisayar bilimlerinde dinamik programlama ile çözülen bazı ünlü problemlere örnek olarak dizi hizalama ve A noktasından B noktasına giden en kısa yolun hesaplanması verilebilir.

Bununla birlikte, durumların boyutu (yani olası konfigürasyonların sayısı) nispeten büyük olduğunda dinamik programlama uygun bir yaklaşım değildir. Bu gibi durumlarda RL, MDP'lerin çözümü için çok daha verimli ve pratik bir alternatif yaklaşım olarak kabul edilir.

Markov karar süreçlerinin matematiksel formülasyonu

Etkileşimli ve sıralı bir karar verme sürecini öğrenmeyi gerektiren, t zaman adımındaki kararın sonraki durumları etkilediği problem türleri, matematiksel olarak MDP'ler olarak biçimlendirilir.

RL'de ajan/çevre etkileşimleri söz konusu olduğunda, ajanın başlangıç durumunu S_0 olarak gösterirsek, ajan ve çevre arasındaki etkileşimler aşağıdaki gibi bir diziyle sonuçlanır:

$$\{S_0, A_0, R_1\}, \{S_1, A_1, R_2\}, \{S_2, A_2, R_3\}, \ldots$$

Parantezlerin sadece görsel bir yardımcı olarak kullanıldığına dikkat ediniz. Burada, S_t ve A_t t zaman adımında alınan durumu ve eylemi temsil eder. R_{t+1} , A_t eylemini gerçekleştirdikten sonra çevreden alınan ödülü gösterir. S_t , R_{t+1} ve A_t 'nin önceden tanımlanmış sonlu kümelerden değerler alan zamana bağlı rastgele değişkenler olduğunu unutmayın
sırasıyla $s \in \mathcal{S}$, $r \in \mathcal{R}$ ve $a \in \mathcal{A}$ ile gösterilir. Bir MDP'de, bu zamana bağlı rastgele değişkenler,
S_t ve R_{t+1} , sadece bir önceki zaman adımı olan t - 1'deki değerlerine bağlı olan olasılık dağılımlarına sahiptir. $S_{t+1} = s'$ ve $R_{t+1} = r$ için olasılık dağılımı, önceki durum (S_t) ve alınan eylem (A_t) üzerinde koşullu bir olasılık olarak aşağıdaki gibi yazılabilir:

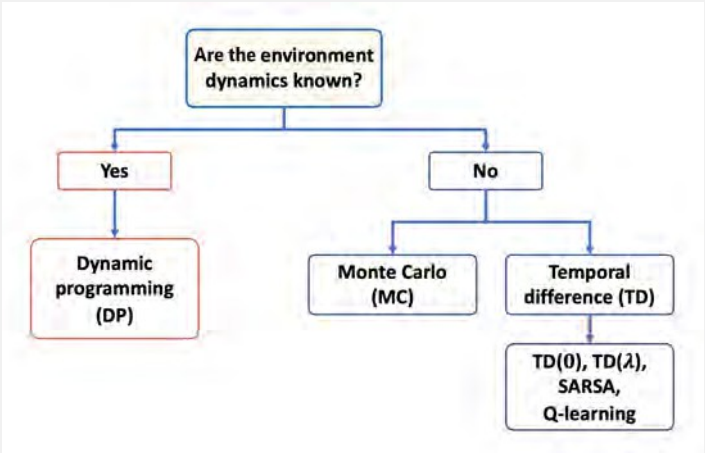
$$p(s', r|s, a) \stackrel{\text{def}}{=} P(S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a)$$

Bu olasılık dağılımı, **ortamın** (veya ortam modelinin) **dinamiklerini** tamamen tanımlar çünkü bu dağılıma dayanarak ortamın tüm geçiş olasılıkları hesaplanabilir. Bu nedenle, ortam dinamikleri farklı RL yöntemlerini kategorize etmek için merkezi bir kriterdir. Ortamın bir modelini gerektiren veya ortamın bir modelini (yani ortam dinamiklerini) öğrenmeye çalışan RL yöntemlerine, *modelsiz* yöntemlerin aksine model *tabanlı yöntemler* denir.

Modelsiz ve model tabanlı RL

Olasılık $p(s', r|s, a)$ bilindiğinde, öğrenme görevi dinamik programlama ile çözülebilir. Ancak birçok gerçek dünya probleminde olduğu gibi çevrenin dinamikleri bilinmediğinde, bilinmeyen çevre dinamiklerini telafi etmek için çevreyle etkileşime girerek çok sayıda örnek elde etmemiz gerekecektir.

Bu sorunla başa çıkmak için iki ana yaklaşım modelsiz **Monte Carlo** (**MC**) ve **zamansal fark** (**TD**) yöntemleridir. Aşağıdaki grafik iki ana kategoriyi ve her bir yöntemin dallarını göstermektedir:



Şekil 19.2: Çevre dinamiklerine bağlı olarak kullanılabilecek farklı modeller

Bu bölümde bu farklı yaklaşımları ve bunların teoriden pratik algoritmalara kadar uzanan dallarını ele alacağız.

Çevre dinamikleri, verilen durumlar için belirli eylemler her zaman yapılıyorsa veya hiç yapılmıyorsa, yani $p(s', r|s, a) \in \{0,1\}$ ise deterministik olarak kabul edilebilir. Aksi takdirde, daha genel durumda, ortam stokastik davranışa sahip olacaktır.

Bu stokastik davranışı anlamlandırmak için, gelecekteki durumu gözlemleme olasılığını göz önünde bulunduralım
$S_{t+1} = s'$ mevcut durum $S_t = s$ ve gerçekleştirilen eylem $A_t = a$ ile koşullandırılmıştır:

$$p(s' \,|ss\ a) \stackrel{\text{def}}{=} P(_{Stt1} = s'_{\,|St} = ss_{\,At} = a)$$

 alinpikbarbell@gmail.com

Olası tüm ödüllerin toplamı alınarak marjinal bir olasılık olarak hesaplanabilir:

$$p(s' \mid s, a) \stackrel{\text{def}}{=} \sum_{r \in \mathcal{R}} p(s', r \mid s, a)$$

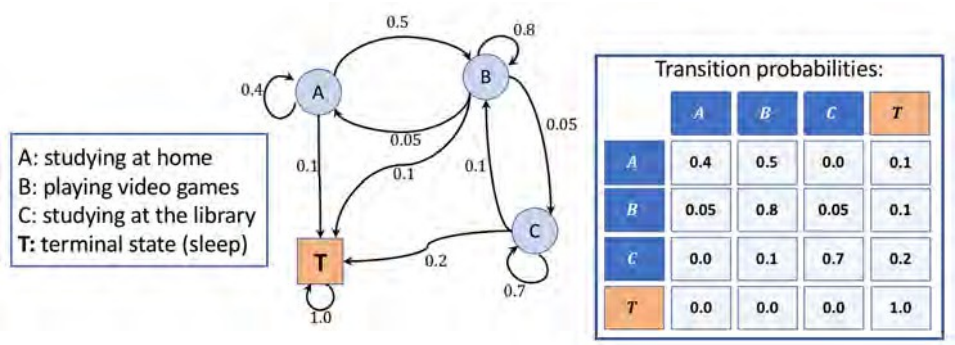
Bu olasılığa **durum-geçiş olasılığı** denir. Durum geçiş olasılığına göre, eğer çevre dinamikleri deterministik ise, o zaman ajan $S_t = s$ durumunda $A_t = a$ eylemini gerçekleştirdiğinde, bir sonraki duruma, $S_{t+1} = s'$, geçiş yüzde 100 kesin olacaktır, yani $p(s' \mid s, a) = 1$ olacaktır.

Bir Markov sürecinin görselleştirilmesi

Bir Markov süreci, grafikteki düğümlerin ortamın farklı durumlarını temsil ettiği yönlendirilmiş döngüsel bir grafik olarak temsil edilebilir. Grafiğin kenarları (yani düğümler arasındaki bağlantılar) durumlar arasındaki geçiş olasılıklarını temsil eder.

Örneğin, bir öğrencinin üç farklı durum arasında karar verdiğini düşünelim: (A) evde sınava çalışmak, (B) evde video oyunları oynamak veya (C) kütüphanede ders çalışmak. Ayrıca, uyumak için bir terminal durum (T) vardır. Kararlar her saat verilir ve bir karar verdikten sonra öğrenci o saat için seçilen durumda kalır. O halde, evde kalırken (A durumu), öğrencinin aktiviteyi video oyunu oynamaya çevirme olasılığının yüzde 50 olduğunu varsayın. Öte yandan, öğrenci B durumundayken (video oyunları oynarken), öğrencinin sonraki saatlerde video oyunları oynamaya devam etme ihtimali nispeten yüksektir (yüzde 80).

Öğrencinin davranışının dinamikleri, döngüsel bir grafik ve bir geçiş tablosu içeren *Şekil 19.3*'te bir Markov süreci olarak gösterilmektedir:



	A	B	C	T
A	0.4	0.5	0.0	0.1
B	0.05	0.8	0.05	0.1
C	0.0	0.1	0.7	0.2
T	0.0	0.0	0.0	1.0

Şekil 19.3: Öğrencinin Markov süreci

Grafiğin kenarlarındaki değerler, öğrencinin davranışının geçiş olasılıklarını temsil eder ve değerleri de sağdaki tabloda gösterilmiştir. Tablodaki satırları değerlendirirken, her durumdan (düğümden) çıkan geçiş olasılıklarının toplamının her zaman 1 olduğuna dikkat ediniz.

Epizodik ve sürekli görevler

Ajan çevre ile etkileşime girdikçe, gözlemler veya durumlar dizisi bir yörünge oluşturur. İki tür yörünge vardır. Bir ajanın yörüngesi, her biri $t = 0$ zamanında başlayacak ve S_T terminal durumunda ($t = T$'de) sona erecek şekilde alt bölümlere ayrılabilirse, görev *epizodik görev* olarak adlandırılır.

Öte yandan, yörünge bir son durum olmaksızın sonsuza kadar *devam* ediyorsa, görev *devam eden bir görev olarak* adlandırılır.

Satranç oyunu için öğrenen bir ajanla ilgili görev epizodik bir görevdir, oysa bir evi düzenli tutan bir temizlik robotu tipik olarak sürekli bir görevi yerine getirmektedir. Bu bölümde sadece epizodik görevleri ele alacağız.

Epizodik görevlerde bir epizot, bir ajanın bir başlangıç durumundan (S_0) bir son duruma (S_T) kadar izlediği bir dizi veya yörüngedir:

$$S_0 , A_0 , R_1 , S_1 , A_1 , R_2 , ..., S_t , A , R_{tt+1} , ..., S_{t-1} , A_{t-1} , R_t , S_t$$

Şekil 19.3'te gösterilen ve bir öğrencinin sınava çalışma görevini tasvir eden Markov süreci için aşağıdaki üç örneğe benzer olaylarla karşılaşabiliriz:

Bölüm 1: *BBCCCCBAT B* p*B*ss (fin*B*l rew*B*rd = +1) Bölüm 2: *ABBBBBT B* f*B*il (fin*B*l rew*B*rd = -1)

Bölüm 3: *BCCCCCTB* p*B*ss (fin*B*l rew*B*rd = +1)

RL terminolojisi: getiri, politika ve değer fonksiyonu

Daha sonra, bu bölümün geri kalanında ihtiyaç duyacağımız RL'ye özgü bazı ek terminolojiyi tanımlayalım.

Geri dönüş

t zamanındaki *getiri,* bir bölümün tüm süresinden elde edilen kümelenmiş ödüldür. R_{t+1} = *r*'nin *t* zamanında bir eylem, A_t , gerçekleştirdikten sonra elde edilen *anlık ödül* olduğunu hatırlayın; *alt-sıralı* ödüller R_{t+2} , R_{t+3} , ve benzerleridir.

Bu durumda t zamanındaki getiri, anlık ödülün yanı sıra sonraki ödüllerden de aşağıdaki şekilde hesaplanabilir:

$$Gt \stackrel{\text{def}}{=} Rtt1 + \gamma\gamma Rtt2 + \gamma\gamma 2Rtt + \cdots = \sum_{kkkk} \gamma\gamma kk Rttkkt1$$

Burada, $\gamma\gamma$ [0, 1] aralığındaki *iskonto faktörüdür*. Parametre $\gamma\gamma$, gelecekteki ödüllerin mevcut anda (*t* zamanı) ne kadar "değerli" olduğunu gösterir. $\gamma\gamma$ $\gamma\gamma$ ayarlayarak, gelecekteki ödülleri önemsemediğimizi ima ettiğimizi unutmayın. Bu durumda, getiri anlık ödüle eşit olacak, *t* + 1'den sonraki ödüller göz ardı edilecek ve temsilci dar görüşlü olacaktır. Öte yandan, $\gamma\gamma$ γ 1 ise, getiri sonraki tüm ödüllerin ağırlıklandırılmamış toplamı olacaktır.

Ayrıca, getiri denkleminin *özyineleme* kullanılarak daha basit bir şekilde ifade edilebileceğini unutmayın aşağıdaki gibi:

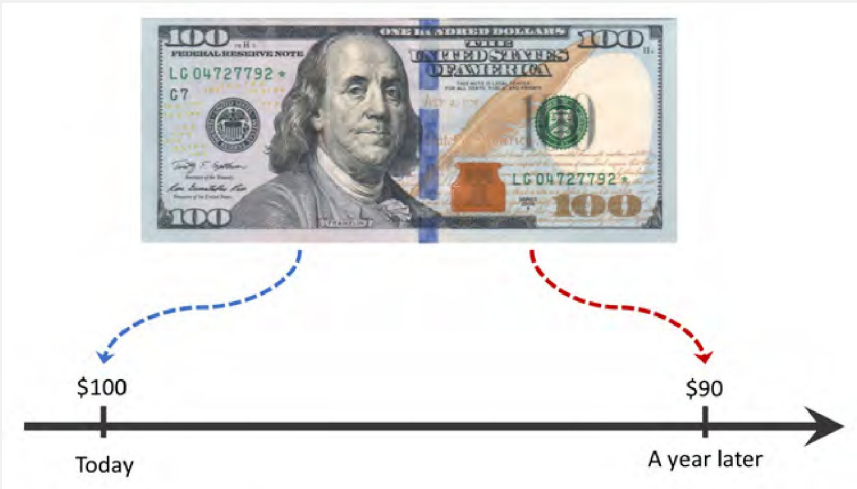
$$Gt = Rtt1 + \gamma\gamma Gtt1 = r + \gamma\gamma Gtt1$$

 alinpikbarbell@gmail.com

Bu, t zamanındaki getirinin r anlık ödül artı $t + 1$ zamanındaki indirgenmiş gelecek getiriye eşit olduğu anlamına gelir. Bu, getiri hesaplamalarını kolaylaştıran çok önemli bir özelliktir.

İskonto faktörünün arkasındaki sezgi

İskonto faktörünü anlamak için, 100$'lık bir banknotu bugün kazanmanın değerini bir yıl sonra kazanmaya kıyasla gösteren *Şekil 19.4'ü* göz önünde bulundurun. Enflasyon gibi belirli ekonomik koşullar altında, bu 100$'ı şu anda kazanmak, gelecekte kazanmaktan daha değerli olabilir:



Şekil 19.4: Zaman içinde 100 dolarlık bir banknotun değerine dayalı bir iskonto faktörü örneği

Dolayısıyla, bu banknot şu anda 100 $ değerindeyse, $\gamma\gamma$ $\gamma\gamma$ iskonto faktörü ile bir yıl sonra 90 $ değerinde olacağını söyleyebiliriz.

Önceki öğrenci örneğimizdeki bölümler için farklı zaman adımlarındaki getiriyi hesaplayalım. $\gamma\gamma$ γ $\gamma\gamma$ ve verilen tek ödülün sınav sonucuna bağlı olduğunu varsayalım (sınavı geçmek için +1, geçememek için -1). Ara zaman adımları için ödüller 0'dır.

Bölüm 1: *BBCCCCBAT B* p*B*ss (fin*B*l rew*B*rd = +1):

- $t\ tt\ t_{G0}\ t_{R1} + \gamma\gamma R2 + \gamma\gamma^2{}_{R3+} \cdots + \gamma\gamma^6{}_{R7}$

  $\rightarrow {}_{G0} = 0 + 0 \times \gamma\gamma + \gamma + 1 \times \gamma\gamma^6 = 0{,}9^6 \approx 0{,}531$
- $t\ t\ 1\ t\ G_1\ t\ 1 \times \gamma\gamma^5\ t\ 0{,}590$
- $t\ tt\ t\ G_2\ t\ 1 \times \gamma\gamma^4\ t\ 0{,}656$
- ...
- $t\ tt\ t\ G_6\ t\ 1 \times \gamma\gamma\ t\ \gamma\gamma\gamma$
- $t\ tt\ t\ G_7\ t\ 1\ t\ 1$

Bölüm 2: *ABBBBBBT A AAiA* (*A*in*AA* rew*A*rd = -1):

- *t tt* $_{G0}$ *t* -1 × $\gamma\gamma^8$ *t -t*.43*t*
- *t t* 1 *t* G_0 *t* -1 × $\gamma\gamma^7$ *t* -0,478
- ...
- ***t*** *tt* ***t*** G_0 ***t*** -1 × $\gamma\gamma$ ***t*** -$\gamma\gamma\gamma$
- *t tt t G* $_{10}$ *t* -1

Üçüncü bölüm için getirilerin hesaplanmasını okuyucuya bir alıştırma olarak bırakıyoruz.

Politika

Tipik olarak $\pi(a|s)$ ile gösterilen bir *politika, bir* sonraki eylemi belirleyen bir fonksiyondur ve deterministik ya da stokastik (yani, bir sonraki eylemi gerçekleştirme olasılığı) olabilir. Stokastik bir politika, bir ajanın belirli bir durumda alabileceği eylemler üzerinde bir olasılık dağılımına sahiptir:

$$\pi(a|s) \stackrel{\text{def}}{=} P[_{At} = {}_{a|St} = s]$$

Öğrenme süreci boyunca, temsilci daha fazla deneyim kazandıkça politika değişebilir. Örneğin, ajan tüm eylemlerin olasılığının tekdüze olduğu rastgele bir politikadan başlayabilir; bu arada, ajan umarız optimal politikaya ulaşmak için politikasını optimize etmeyi öğrenecektir. *Optimal politika* π_* $(a|s)$ en yüksek getiriyi sağlayan politikadır.

Değer fonksiyonu

Durum-değer fonksiyonu olarak da adlandırılan *değer fonksiyonu*, her bir durumun *iyiliğini, diğer bir* deyişle belirli bir durumda olmanın ne kadar iyi veya kötü olduğunu ölçer. İyilik kriterinin getiriye dayandığını unutmayın.

Şimdi, G_t getirisine dayanarak, s durumunun değer fonksiyonunu π *politikasını izledikten* sonra beklenen getiri (tüm olası dönemler üzerindeki ortalama getiri) olarak tanımlıyoruz:

$$v\pi(s) \stackrel{\text{def}}{=} E\pi[_{Gt|St} = s] = {}_{E\pi}\,[\sum_{kkkk} \gamma\gamma^{kkk-1}\, Rtkk1|\,_{St} = ss$$

Gerçek bir uygulamada, değer fonksiyonunu genellikle arama tabloları kullanarak tahmin ederiz, böylece birden çok kez yeniden hesaplamak zorunda kalmayız. (Bu dinamik programlama yönüdür.) Örneğin, pratikte, bu tür tablo yöntemlerini kullanarak değer fonksiyonunu tahmin ettiğimizde, tüm durum değerlerini $V(s)$ ile gösterilen bir tabloda *saklarız.* Bir Python uygulamasında bu, indisleri farklı durumları ifade eden bir liste veya NumPy dizisi olabilir; veya sözlük anahtarlarının durumları ilgili değerlerle eşlediği bir Python sözlüğü olabilir.

Ayrıca, her durum-eylem çifti için *eylem-değer fonksiyonu* olarak adlandırılan ve q_π $(ss\ a)$ ile gösterilen bir değer de tanımlayabiliriz. Eylem-değer fonksiyonu, temsilci $S_t = s$ durumundayken ve $A_t = a$ eylemini gerçekleştirdiğinde beklenen getiriyi G_t ifade eder.

 alinpikbarbell@gmail.com

Durum-değer fonksiyonunun tanımını durum-eylem çiftlerine genişleterek aşağıdakileri elde ederiz:

$$q\pi(ss\ a) \stackrel{\text{def}}{=} E\pi[_{Gt|St} = ss\ _{At} = a] = {}_{E\pi}[\sum_{kkkk} \gamma\gamma^{kkk-1}\ Rtkkk1|\ _{St} = ss\ _{At} = aa$$

Bu, optimal politikaya $\pi_*(a|s)$, $v_*(s)$ ve $q_*(ss\ a)$ olarak atıfta bulunmaya benzer şekilde optimal durum-değer ve eylem-değer fonksiyonlarını da ifade eder.

Değer fonksiyonunun tahmin edilmesi RL yöntemlerinin önemli bir bileşenidir. Bu bölümün ilerleyen kısımlarında durum-değer fonksiyonunu ve eylem-değer fonksiyonunu hesaplamanın ve tahmin etmenin farklı yollarını ele alacağız.

Ödül, getiri ve değer fonksiyonu arasındaki fark

Ödül, çevrenin mevcut durumu göz önüne alındığında ajanın bir eylemde bulunmasının bir sonucudur. Başka bir deyişle ödül, bir durumdan diğerine geçmek için bir eylem gerçekleştirdiğinde ajanın aldığı bir sinyaldir. Ancak, her eylemin olumlu ya da olumsuz bir ödül getirmediğini unutmayın; satranç örneğimizi düşünün; olumlu bir ödül yalnızca oyunu kazandığınızda alınır ve tüm ara eylemler için ödül sıfırdır.

Bir durumun kendisinin, bu durumun ne kadar iyi veya kötü olduğunu ölçmek için ona atadığımız belirli bir değeri vardır - *değer fonksiyonu* burada devreye girer. Tipik olarak, "yüksek" veya "iyi" değere sahip durumlar, beklenen *getirisi yüksek* olan ve belirli bir politika uygulandığında muhtemelen yüksek bir ödül getirecek olan durumlardır.

Örneğin, bir kez daha satranç oynayan bir bilgisayarı ele alalım. Oyunun sonunda olumlu bir ödül ancak bilgisayar oyunu kazanırsa verilebilir. Bilgisayar oyunu kaybederse (pozitif) bir ödül yoktur. Şimdi, bilgisayarın rakibin vezirini ele geçiren belirli bir satranç hamlesini bilgisayar için herhangi bir olumsuz sonuç doğurmadan gerçekleştirdiğini düşünün. Bilgisayar sadece oyunu kazandığında bir ödül alacağından, rakibin vezirini ele geçiren bu hamleyi yaparak anında bir ödül almaz. Ancak, yeni durum (veziri ele geçirdikten sonra tahtanın durumu) yüksek bir değere sahip olabilir ve bu da bir ödül getirebilir (eğer oyun daha sonra kazanılırsa). Sezgisel olarak, rakibin vezirini ele geçirmeyle ilişkili yüksek değerin, veziri ele geçirmenin genellikle oyunu kazanmakla sonuçlanmasıyla ve dolayısıyla yüksek beklenen getiri veya değerle ilişkili olduğunu söyleyebiliriz. Ancak, rakibin vezirini ele geçirmenin her zaman oyunu kazanmaya yol açmadığını unutmayın; dolayısıyla, temsilcinin pozitif bir ödül alması muhtemeldir, ancak bu garanti değildir.

Kısacası, getiri tüm bir bölüm için ödüllerin ağırlıklı toplamıdır, bu da satranç örneğimizde iskonto edilmiş nihai ödüle eşit olacaktır (çünkü sadece bir ödül vardır). Değer fonksiyonu, tüm olası bölümler üzerindeki beklentidir ve temel olarak belirli bir hamleyi yapmanın ortalama olarak ne kadar "değerli" olduğunu hesaplar.

Doğrudan bazı RL algoritmalarına geçmeden önce, politika değerlendirmesini uygulamak için kullanabileceğimiz Bellman denkleminin türetilmesini kısaca gözden geçirelim.

Bellman denklemini kullanan dinamik programlama

Bellman denklemi, birçok RL algoritmasının temel unsurlarından biridir. Bellman denklemi, değer fonksiyonunun hesaplanmasını basitleştirir, öyle ki birden fazla zaman adımı üzerinde toplama yapmak yerine, getiriyi hesaplamak için kullanılan özyinelemeye benzer bir özyineleme kullanır.

Toplam getiri $G_t = r + rrG_{tt1}$ için özyinelemeli denkleme dayanarak, değer fonksiyonunu aşağıdaki gibi yeniden yazabiliriz:

$$\begin{aligned} v\pi(s) &\stackrel{\text{def}}{=} E\pi[_{Gt|St} = s] \\ &= E\pi[r + _{rrGtt1|St} = s] \\ &= r + rr\, E\pi[_{Gtt1|St} = s] \end{aligned}$$

Anlık ödül r'nin t zamanında sabit ve bilinen bir miktar olduğu için beklentiden çıkarıldığına dikkat edin.

Benzer şekilde, eylem-değer fonksiyonu için de şöyle yazabiliriz:

$$\begin{aligned} q\pi(ss\ a) &\stackrel{\text{def}}{=} _{E\pi}[_{Gt|St} = ss\ _{At} = a] \\ &= E\pi[r + _{rrGtt1|St} = ss\ _{At} = a] \\ &= r + rr\, E\pi[_{Gtt1|St} = ss\ _{At} = a] \end{aligned}$$

Bir sonraki s' durumunun tüm olasılıklarını ve karşılık gelen r ödüllerini toplayarak beklentiyi hesaplamak için çevre dinamiklerini kullanabiliriz:

$$v\pi(s) = \sum_{\pi(a|s)} \sum_{sFaS,raR} p(s', r|s, a)[r + rrE\pi[_{Gtt1|Stt1} = s']]$$

Şimdi, getiri beklentisinin, $E_\pi[G_{tt1}|S_t = s']$, esasen durum-değer fonksiyonu olduğunu görebiliriz $_\pi v(s')$. Dolayısıyla, $v_\pi(s)$ 'yi $v_\pi(s')$ 'nin bir fonksiyonu olarak yazabiliriz:

$$v\pi(s) = \sum_{\pi(a|s)} \sum_{sFaS,raR} p(s', r|s, a)[r + rr\, v\pi(s')]$$

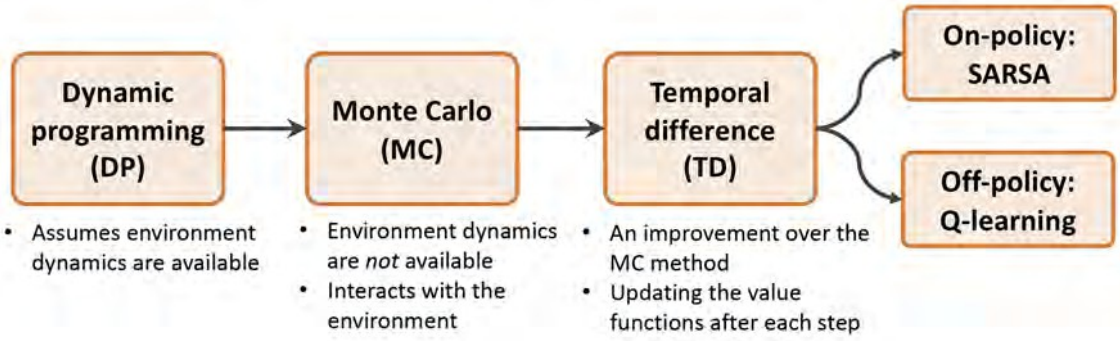
Buna **Bellman denklemi** denir ve bir s durumunun değer fonksiyonunu bir sonraki s' *durumunun* değer fonksiyonu ile ilişkilendirir. Bu, değer fonksiyonunun hesaplanmasını büyük ölçüde basitleştirir çünkü zaman ekseni boyunca yinelemeli döngüyü ortadan kaldırır.

Takviyeli öğrenme algoritmaları

Bu bölümde, bir dizi öğrenme algoritmasını ele alacağız. Geçiş dinamiklerinin ya da ortam dinamiklerinin, yani $p(s', r|s, a)$- bilindiğini varsayan dinamik programlama ile başlayacağız. Ancak, çoğu RL probleminde durum böyle değildir. Bilinmeyen çevre dinamiklerini aşmak için, çevre ile etkileşime girerek öğrenen RL teknikleri geliştirilmiştir. Bu teknikler arasında **Monte Carlo** (**MC**), **zamansal fark** (**TD) öğrenimi ve** giderek daha popüler hale gelen Q-öğrenme ve derin Q-öğrenme yaklaşımları yer almaktadır.

alinpikbarbell@gmail.com

Şekil 19.5, dinamik programlamadan Q-öğrenmeye kadar ilerleyen RL algoritmalarının seyrini açıklamaktadır:



Şekil 19.5: Farklı RL algoritma türleri

Bu bölümün ilerleyen kısımlarında, bu RL algoritmalarının her birini adım adım inceleyeceğiz. MC'ye geçmeden önce dinamik programlama ile başlayacağız ve son olarak TD ve onun dalları olan politika içi **SARSA** (**durum-eylem-ödül-durum-eylem**) ve politika dışı Q-öğrenmeye geçeceğiz. Bazı pratik modeller oluştururken derin Q-öğrenmeye de geçeceğiz.

Dinamik programlama

Bu bölümde, aşağıdaki varsayımlar altında RL problemlerini çözmeye odaklanacağız:

- Çevre dinamikleri hakkında tam bilgiye sahibiz; yani, tüm geçiş olasılıkları $p(s', r|s, a)$- bilinmektedir.
- Temsilcinin durumu Markov özelliğine sahiptir, yani bir sonraki eylem ve ödül yalnızca mevcut duruma ve bu anda veya mevcut zaman adımında yaptığımız eylem seçimine bağlıdır.

Markov karar süreci (**MDP)** kullanan RL problemleri için matematiksel formülasyon bu bölümün başlarında tanıtılmıştır. Tekrar hatırlatmaya ihtiyaç duyarsanız, lütfen π politikasını takip eden $v_\pi(s)$ değer fonksiyonunun ve çevre dinamikleri kullanılarak türetilen Bellman denkleminin resmi tanımını tanıtan *Markov karar süreçlerinin matematiksel* formülasyonu başlıklı bölüme bakın.

Dinamik programlamanın RL problemlerini çözmek için pratik bir yaklaşım olmadığını vurgulamalıyız. Dinamik programlamanın kullanılmasındaki sorun, ortam dinamikleri hakkında tam bilgi sahibi olunduğunu varsaymasıdır ki bu da çoğu gerçek dünya uygulaması için genellikle mantıksız veya pratik değildir. Bununla birlikte, eğitim açısından bakıldığında, dinamik programlama RL'nin basit bir şekilde tanıtılmasına yardımcı olur ve daha gelişmiş ve karmaşık RL algoritmalarının kullanımını motive eder.

Aşağıdaki alt bölümlerde açıklanan görevler aracılığıyla iki ana hedef vardır:

1. Gerçek durum-değer fonksiyonunu elde edin, $v_\pi(s)$; bu görev tahmin görevi olarak da bilinir ve *politika değerlendirmesi* ile gerçekleştirilir.
2. Optimal değer fonksiyonunu bulun, $v_*(s)$, bu da *genelleştirilmiş politika iterasyonu* ile gerçekleştirilir.

Politika değerlendirmesi - dinamik programlama ile değer fonksiyonunun tahmin edilmesi

Bellman denklemine dayanarak, çevre dinamikleri bilindiğinde dinamik programlama ile keyfi bir politika π için değer fonksiyonunu hesaplayabiliriz. Bu değer fonksiyonunu hesaplamak için, her durum için sıfır değerle başlatılan $v^{\langle 0\rangle}(s)$'den başladığımız iteratif bir çözümü uyarlayabiliriz. Daha sonra, her $i + 1$ iterasyonunda, Bellman denklemine dayalı olarak her bir durum için değerleri güncelleriz, bu da bir önceki iterasyondaki durumların değerlerine dayanır, *i,* aşağıdaki gibi:

$$v^{\langle i+1\rangle}(s) = \sum_{a} \pi(a|s) \sum_{s' \in S, r \in R} p(s', r|s, a)[r + \gamma v^{\langle i\rangle}(s')]$$

Yinelemeler sonsuza doğru arttıkça, $v^{\langle i\rangle}(s)$'nin gerçek durum-değer fonksiyonu olan $v_\pi(s)$'ye yakınsadığı gösterilebilir.

Ayrıca, burada çevre ile etkileşime girmemize gerek olmadığına dikkat edin. Bunun nedeni, çevre dinamiklerini zaten doğru bir şekilde biliyor olmamızdır. Sonuç olarak, bu bilgiden faydalanabilir ve değer fonksiyonunu kolayca tahmin edebiliriz.

Değer fonksiyonunu hesapladıktan sonra, politikamız hala rastgele bir politika ise, bu değer fonksiyonunun bizim için nasıl yararlı olabileceği açık bir sorudur. Cevap, aslında bu hesaplanan $v_\pi(s)$ değerini, daha sonra göreceğimiz gibi, politikamızı iyileştirmek için kullanabileceğimizdir.

Tahmin edilen değer fonksiyonunu kullanarak politikanın iyileştirilmesi

Mevcut politika π'yi takip ederek $v_\pi(s)$ değer fonksiyonunu hesapladığımıza göre, $v_\pi(s)$'yi kullanmak ve mevcut politika π'yi iyileştirmek istiyoruz. Bu, her s durumu için π' politikasını izleyerek mevcut π politikasını kullanmaktan daha yüksek veya en azından eşit değer sağlayacak yeni bir politika, π', bulmak istediğimiz anlamına gelir. Matematiksel olarak, iyileştirilmiş politika için bu hedefi, π', olarak ifade edebiliriz:

$$v_{\pi'}(s) \geq v_\pi(s) \ \forall s \in S$$

İlk olarak, bir politikanın, π, ajan s durumundayken her bir eylemi, a, seçme olasılığını belirlediğini hatırlayın. Şimdi, her durum için her zaman daha iyi veya eşit değere sahip π' bulmak için, ilk olarak $v_\pi(s)$ değer fonksiyonunu kullanarak hesaplanan durum değerine dayalı olarak her durum, s ve eylem, *a için eylem-değer* fonksiyonunu, $q_\pi(s, a)$, hesaplarız. Tüm durumlar arasında yineleme yaparız ve her s durumu için, a eylemi seçildiğinde oluşacak bir sonraki s' *durumunun* değerini karşılaştırırız.

Tüm durum-eylem çiftlerini $q_\pi(s, a)$ üzerinden değerlendirerek en yüksek durum değerini elde ettikten sonra, ilgili eylemi mevcut politika tarafından seçilen eylemle karşılaştırabiliriz. Mevcut politika tarafından önerilen eylem (yani $\arg\max_a \pi(a|s)$), mevcut politika tarafından önerilen eylemden farklıysa eylem-değer fonksiyonu (yani, $\arg\max_a q_\pi(s, a)$), o zaman politikayı yeniden atayarak güncelleyebiliriz en yüksek eylem değerini veren eylemle eşleşecek eylem olasılıkları, $q_\pi(s, a)$. Buna şu ad verilir

alinpikbarbell@gmail.com

politika iyileştirme algoritması.

Politika yinelemesi

Önceki alt bölümde açıklanan politika iyileştirme algoritması kullanılarak, mevcut politika zaten optimal olmadığı sürece (yani her s s S için $v_\pi(s) = v_{\pi F}(s) = v_*(s)$) politika iyileştirmenin kesinlikle daha iyi bir politika sağlayacağı gösterilebilir. Bu nedenle, politika değerlendirmesini ve ardından politika iyileştirmesini iteratif olarak gerçekleştirirsek, en uygun politikayı bulmamız garanti edilir.

Bu tekniğin **genelleştirilmiş politika yinelemesi (GPI) olarak adlandırıldığını ve** birçok RL yöntemi arasında yaygın olduğunu unutmayın. GPI'yı bu bölümün ilerleyen kısımlarında MC ve TD öğrenme yöntemleri için kullanacağız.

Değer yinelemesi

Politika değerlendirme ($v_\pi(s)$ ve $q_\pi(ss\ a)$ hesaplama) ve politika iyileştirme ($v\pi F(s) \geq v\pi(s)\ \forall s\ s\ S$ olacak şekilde π' bulma) işlemlerini tekrarlayarak optimal politikaya ulaşabileceğimizi gördük. Bununla birlikte, politika değerlendirme ve politika iyileştirme görevlerini tek bir adımda birleştirirsek daha verimli olabilir.

Aşağıdaki denklem, bir sonraki durum değeri ile anlık ödülünün ağırlıklı toplamını ($r + rv^{\langle i \rangle}(s')$) maksimize eden eyleme dayalı olarak $i + 1$ iterasyonu için değer fonksiyonunu günceller ($v^{\langle ii1 \rangle}$ ile gösterilir):

$$^{(i\text{ƶ}1)}v(s) = \underset{a}{\text{maks}} \sum_{sF,r} p(s', r|s, a)[r + r\, v^{(i)}(s')]$$

Bu durumda, $v^{(i\text{ƶ}1)}(s)$ için güncellenmiş değer, tüm olası eylemler arasından en iyi eylem seçilerek maksimize edilirken, politika değerlendirmesinde güncellenmiş değer, tüm eylemler üzerinden ağırlıklı toplamı kullanıyordu.

Durum-değer ve eylem-değer fonksiyonlarının tablo tahminleri için gösterim

Çoğu RL literatüründe ve ders kitaplarında, matematiksel fonksiyonlar olarak sırasıyla gerçek durum-değer ve gerçek eylem-değer fonksiyonlarını ifade etmek için küçük harf v_π ve q_π kullanılır.

Öte yandan, pratik uygulamalar için bu değer fonksiyonları arama tabloları olarak tanımlanır. Bu değer fonksiyonlarının tablo tahminleri $V(S_t = s) \approx v_\pi(s)$ ve $QQ_\pi(S_t = ss\ A_t = a) \approx q_\pi(ss\ a)$ ile gösterilir. Bu bölümde de bu notasyonu kullanacağız.

Monte Carlo ile pekiştirmeli öğrenme

Önceki bölümde gördüğümüz gibi, dinamik programlama ortamın dinamiklerinin tamamen bilindiğine dair basit bir varsayıma dayanmaktadır. Dinamik programlama yaklaşımından uzaklaşarak, şimdi çevre dinamikleri hakkında herhangi bir bilgiye sahip olmadığımızı varsayıyoruz.

Yani, çevrenin durum-geçiş olasılıklarını bilmiyoruz ve bunun yerine ajanın çevreyle *etkileşime* girerek öğrenmesini istiyoruz. MC yöntemleri kullanıldığında, öğrenme süreci *simüle edilmiş deneyim olarak* adlandırılan sürece dayanır.

 alinpikbarbell@gmail.com

MC tabanlı RL için, olasılıksal bir politika izleyen bir ajan sınıfı tanımlarız, π, ve bu politikaya dayanarak, ajanımız her adımda bir eylem gerçekleştirir. Bu, simüle edilmiş bir bölümle sonuçlanır.

Daha önce, bir durumun değerinin o durumdan beklenen getiriyi göstermesi için durum-değer fonksiyonunu tanımlamıştık. Dinamik programlamada, bu hesaplama ortam dinamiklerinin, yani $p(s', r|s, a)$ bilgisine dayanıyordu.

Ancak bundan sonra çevre dinamiklerine ihtiyaç duymayan algoritmalar geliştireceğiz. MC tabanlı yöntemler, bir ajanın çevreyle etkileşime girdiği simüle edilmiş bölümler oluşturarak bu sorunu çözer. Bu simüle edilmiş bölümlerden, o simüle edilmiş bölümde ziyaret edilen her durum için ortalama getiriyi hesaplayabileceğiz.

MC kullanarak durum-değer fonksiyonu tahmini

Bir dizi bölüm oluşturulduktan sonra, her bir *s durumu* için, *s* durumunun değerini hesaplamak üzere tümü s durumundan geçen bölümler kümesi dikkate alınır. Değer fonksiyonuna karşılık gelen değeri elde etmek için bir arama tablosu kullanıldığını varsayalım, $V(S_t = s)$. Değer fonksiyonunu tahmin etmek için MC güncellemeleri, *s durumunun* ilk ziyaret edildiği andan başlayarak o bölümde elde edilen toplam getiriye dayanır. Bu algoritma *ilk ziyaret Monte Carlo* değer tahmini olarak adlandırılır.

MC kullanarak eylem-değer fonksiyonu tahmini

Ortam dinamikleri bilindiğinde, *Dinamik programlama* bölümünde gösterildiği gibi, maksimum değeri veren eylemi bulmak için bir adım ileriye bakarak bir durum-değer fonksiyonundan eylem-değer fonksiyonunu kolayca çıkarabiliriz. Ancak, çevre dinamikleri bilinmiyorsa bu mümkün değildir.

Bu sorunu çözmek için, ilk ziyaret MC durum-değer tahminini tahmin etmeye yönelik algoritmayı genişletebiliriz. Örneğin, eylem-değer fonksiyonunu kullanarak her bir durum-eylem çifti için *tahmini* getiriyi hesaplayabiliriz. Bu tahmini getiriyi elde etmek için, *s* durumunu ziyaret etmek ve *a* eylemini gerçekleştirmek anlamına gelen her bir durum-eylem çiftine (*s*, *a*) yapılan ziyaretleri dikkate alırız.

Ancak, bazı eylemler hiçbir zaman seçilemeyeceği için bir sorun ortaya çıkar ve bu da yetersiz keşifle sonuçlanır. Bunu çözmenin birkaç yolu vardır. En basit yaklaşım, her durum-eylem çiftinin bölümün başında sıfır olmayan bir olasılığa sahip olduğunu varsayan *keşifsel başlangıç* olarak adlandırılır.

Bu keşif eksikliği sorunuyla başa çıkmak için bir başka yaklaşım, politika iyileştirme ile ilgili bir sonraki bölümde tartışılacak olan *ϵ-greedy politikası olarak* adlandırılır.

MC kontrolü kullanarak en uygun politikayı bulma

MC kontrolü, bir politikayı iyileştirmek için optimizasyon prosedürünü ifade eder. Önceki bölümdeki (*Dinamik programlama*) politika yineleme yaklaşımına benzer şekilde, optimum politikaya ulaşana kadar politika değerlendirme ve politika iyileştirme arasında tekrar tekrar geçiş yapabiliriz. Dolayısıyla, rastgele bir politikadan başlayarak, π_0 , politika değerlendirme ve politika iyileştirme arasında geçiş yapma süreci aşağıdaki gibi gösterilebilir:

$\pi_0 \xrightarrow{\text{Improve}} \pi_1 \xrightarrow{\text{Improve}} \pi_2 \xrightarrow{\text{Improve}}$

$\pi 0 \rightarrow q_{\pi 0} \rightarrow \pi 1 \rightarrow q_{\pi 1} \rightarrow \pi 2 \rightarrow q_{*} \rightarrow \pi_{*}$.

 olimpikbarbell@gmail.com

Politika iyileştirme - eylem-değer fonksiyonundan açgözlü politikanın hesaplanması

Bir eylem-değer fonksiyonu $q(s, a)$ verildiğinde, aşağıdaki gibi açgözlü (deterministik) bir politika oluşturabiliriz:

$$\pi(s) \stackrel{\text{def}}{=} \arg\max_{a} q(s, a)$$

Keşif eksikliği sorunundan kaçınmak ve daha önce tartışıldığı gibi ziyaret edilmeyen durum-eylem çiftlerini göz önünde bulundurmak için, optimal olmayan eylemlerin küçük bir seçilme şansına (ϵ) sahip olmasına izin verebiliriz. Buna ϵ denir

s durumundaki tüm optimal olmayan eylemlerin minimum $\frac{\epsilon}{|A(s)|}$ olasılığına sahip olduğu -greedy politikası

(0 yerine) ve optimal eylem $1 - \frac{(|A(s)|-1)\times\epsilon}{|A(s)|}$ (1 yerine) olasılığına sahiptir.

Zamansal fark öğrenimi

Şimdiye kadar iki temel RL tekniği gördük-dinamik programlama ve MC tabanlı öğrenme. Dinamik programlamanın çevre dinamikleri hakkında tam ve doğru bilgiye dayandığını hatırlayın. MC tabanlı yöntem ise simüle edilmiş deneyimle öğrenir. Bu bölümde, MC tabanlı RL yaklaşımının bir iyileştirmesi veya uzantısı olarak düşünülebilecek *TD öğrenme* adı verilen üçüncü bir RL yöntemini tanıtacağız.

MC tekniğine benzer şekilde, TD öğrenme de deneyimle öğrenmeye dayanır ve bu nedenle ortam dinamikleri ve geçiş olasılıkları hakkında herhangi bir bilgi gerektirmez. TD ve MC teknikleri arasındaki temel fark, MC'de toplam getiriyi hesaplayabilmek için bölümün sonuna kadar beklemek zorunda olmamızdır.

Ancak TD öğrenmede, bölümün sonuna ulaşmadan önce tahmini değerleri güncellemek için öğrenilen bazı özelliklerden yararlanabiliriz. Buna *bootstrapping* denir (RL bağlamında *bootstrapping* terimi, *Bölüm 7, Topluluk Öğrenmesi için Farklı Modelleri Birleştirme*'de kullandığımız bootstrap tahminleriyle karıştırılmamalıdır).

Dinamik programlama yaklaşımına ve MC tabanlı öğrenmeye benzer şekilde, iki görevi ele alacağız: değer fonksiyonunun tahmin edilmesi (buna değer tahmini de denir) ve politikanın iyileştirilmesi (buna kontrol görevi de denir).

TD tahmini

İlk olarak MC tarafından yapılan değer tahminini tekrar gözden geçirelim. Her bölümün sonunda, her t zaman adımı için getiriyi, G_t , tahmin edebiliyoruz. Bu nedenle, ziyaret edilen durumlar için tahminlerimizi aşağıdaki gibi güncelleyebiliriz:

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t))$$

Burada, G_t tahmini değerleri güncellemek için *hedef getiri* olarak kullanılır ve $(G_t - V(S_t))$ $V(S_t)$ değerine ilişkin mevcut tahminimize eklenen bir *düzeltme* terimidir. α değeri, öğrenme sırasında sabit tutulan öğrenme oranını ifade eden bir hiperparametredir.

MC'de düzeltme teriminin, bölümün sonuna kadar bilinmeyen *gerçek* getiriyi, G_t , kullandığına dikkat edin. Bunu açıklığa kavuşturmak için, gerçek getiriyi, G_t , $G_{t:T}$ olarak yeniden adlandırabiliriz; burada $t{:}T$ alt simgesi, bunun t zaman adımından son zaman adımı olan *T'ye* kadar meydana gelen tüm olayları dikkate alırken t zaman adımında elde edilen getiri olduğunu gösterir.

TD öğrenmede, gerçek getiriyi, $G_{t:T}$, yeni bir hedef getiri ile değiştiririz, $G_{t:t+1}$, bu da değer fonksiyonu, $V(S_t)$ için güncellemeleri önemli ölçüde basitleştirir. TD öğrenmeye dayalı güncelleme formülü aşağıdaki gibidir:

$$V(S_t) \leftarrow V(S_t) + \alpha(G_{tttt1} - V(S_t))$$

Burada hedef getiri, $G_{tttt1} \stackrel{\text{def}}{=} R_{tt1} + \gamma\gamma V(S_{tt1}) = r + \gamma\gamma V(S_{tt1})$, gözlenen ödül, R_{t+1} ve bir sonraki acil adımın tahmini değerini kullanır. MC ve TD arasındaki farka dikkat edin. MC'de, $G_{t:T}$ bölümün sonuna kadar mevcut değildir, bu nedenle oraya ulaşmak için gerektiği kadar adım atmalıyız. Aksine, TD'de hedef getiriyi elde etmek için yalnızca bir adım ileri gitmemiz gerekir. Bu aynı zamanda TD(0) olarak da bilinir.

Ayrıca, TD(0) algoritması, daha fazla gelecek adımı, daha doğrusu n gelecek adımın ağırlıklı toplamını içeren *n-adımlı TD* algoritması olarak adlandırılan algoritmaya genelleştirilebilir. Eğer n = 1 olarak tanımlanırsa, n-adımlı TD prosedürü bir önceki paragrafta açıklanan TD(0) ile aynıdır. Eğer n nn, ancak n-adımlı TD algoritması MC algoritması ile aynı olacaktır. n-adımlı TD için güncelleme kuralı aşağıdaki gibidir:

$$V(S_t) \leftarrow V(St) + \alpha(G_{tttn} - V(S_t))$$

Ve $G_{t:t+n}$ olarak tanımlanır:

$$G_{tttn} \stackrel{\text{def}}{=} \begin{cases} R_{tt1} + \gamma\gamma R_{tt2} + \cdots + \gamma\gamma^{nn1} R_{ttn} + \gamma\gamma^{n} V(S_{ttn}) & \text{eğer } t + n\ t\ T \\ G_{ttT} & \text{Aksi takdirde} \end{cases}$$

MC'ye karşı TD: hangi yöntem daha hızlı yakınsar?

Bu sorunun kesin cevabı hala bilinmemekle birlikte, pratikte TD'nin MC'den daha hızlı yakınsayabildiği deneysel olarak gösterilmiştir. Eğer ilgileniyorsanız, MC ve TD'nin yakınsamaları hakkında daha fazla bilgiyi *Reinforcement Learning* adlı kitapta bulabilirsiniz: *An Introduction, Richard S. Sutton* ve *Andrew G. Barto* tarafından yazılmıştır.

Artık TD algoritmasını kullanarak tahmin görevini ele aldığımıza göre, kontrol görevine geçebiliriz. TD kontrolü için iki algoritmayı ele alacağız: bir *politika içi kontrol* ve bir *politika dışı* kontrol. Her iki durumda da, hem dinamik programlama hem de MC algoritmalarında kullanılan GPI'yı kullanıyoruz. Politika içi TD kontrolünde değer fonksiyonu, ajanın takip ettiği aynı politikadaki eylemlere dayalı olarak güncellenirken; politika dışı algoritmada değer fonksiyonu mevcut politikanın dışındaki eylemlere dayalı olarak güncellenir.

 olimpikbarbell@gmail.com

Politika üzerinde TD kontrolü (SARSA)

Basitlik açısından, sadece tek adımlı TD algoritmasını veya TD(0)'ı ele alıyoruz. Ancak, on-policy TD kontrol algoritması kolaylıkla n-adımlı TD'ye genelleştirilebilir. Durum-değer fonksiyonunu tanımlamak için kullanılan tahmin formülünü eylem-değer fonksiyonunu tanımlamak için genişleterek başlayacağız. Bunu yapmak için, her bir durum-eylem çifti için eylem-değer fonksiyonunu temsil eden bir arama tablosu, yani $Q(S_t , A_t)$ tablosal 2D dizisi kullanacağız. Bu durumda, aşağıdakilere sahip olacağız:

$$QQ(_{St}, _{At}) \leftarrow QQ(_{St}, _{At}) + \alpha[_{Rtt1} + \gamma\gamma\, QQ(Stt1, _{Att1}) - QQ(_{St}, _{At})]$$

Bu algoritma, güncelleme formülünde kullanılan beşliye $(S_t , A_t , R_{t+1} , S_{t+1} , A_{t+1})$ atıfta bulunarak genellikle SARSA olarak adlandırılır.

Dinamik programlama ve MC algoritmalarını açıklayan önceki bölümlerde gördüğümüz gibi, GPI çerçevesini kullanabilir ve rastgele politikadan başlayarak, mevcut politika için eylem-değer fonksiyonunu tekrar tekrar tahmin edebilir ve ardından mevcut eylem-değer fonksiyonuna dayalı $\epsilon\epsilon$-greedy politikasını kullanarak politikayı optimize edebiliriz.

Politika dışı TD kontrolü (Q-öğrenme)

Önceki politika üzerinde TD kontrol algoritmasını kullanırken, eylem-değer fonksiyonunu nasıl tahmin ettiğimizin simüle edilen bölümde kullanılan politikaya dayandığını gördük. Eylem-değer fonksiyonunu güncelledikten sonra, daha yüksek değere sahip olan eylemi alarak politika iyileştirme için ayrı bir adım gerçekleştirilir.

Alternatif (ve daha iyi) bir yaklaşım ise bu iki adımı birleştirmektir. Başka bir deyişle, ajanın π politikasını izlediğini ve mevcut geçiş beşlisi $(S_t , A_t , R_{t+1} , S_{t+1} , A_{t+1})$ ile bir bölüm oluşturduğunu düşünün. Eylem-değer fonksiyonunu ajan tarafından alınan A_{t+1} eylem değerini kullanarak güncellemek yerine, mevcut politikayı izleyen ajan tarafından gerçekten seçilmemiş olsa bile en iyi eylemi bulabiliriz. (Bu yüzden bu bir *politika dışı* algoritma olarak kabul edilir).

Bunu yapmak için, güncelleme kuralını bir sonraki acil durumdaki farklı eylemleri değiştirerek maksimum Q değerini dikkate alacak şekilde değiştirebiliriz. Q-değerlerinin güncellenmesi için değiştirilmiş denklem aşağıdaki gibidir:

$$QQ(_{St}, _{At}) \leftarrow QQ(_{St}, _{At}) + \alpha[_{Rtt1} + \gamma\gamma\ \gamma\gamma\ \underset{a}{QQ}(_{Stt1}, a) - QQ(_{St}, _{At})]$$

Buradaki güncelleme kuralını SARSA algoritmasınınkiyle karşılaştırmanızı öneririz. Gördüğünüz gibi, bir sonraki durumdaki en iyi eylemi buluyoruz, S_{t+1} , ve bunu $Q(S_t , A_t)$ tahminimizi güncellemek için düzeltme teriminde kullanıyoruz.

Bu materyalleri bir perspektife oturtmak için, bir sonraki bölümde, bir *ızgara dünya problemini* çözmek için Q-öğrenme algoritmasının nasıl uygulanacağını göreceğiz.

İlk RL algoritmamızın uygulanması

Bu bölümde, bir *ızgara dünya problemini* çözmek için Q-öğrenme algoritmasının uygulanmasını ele alacağız (bir ızgara dünya, ajanın mümkün olduğunca çok ödül toplamak için dört yönde hareket ettiği iki boyutlu, hücre tabanlı bir ortamdır). Bunu yapmak için OpenAI Gym araç setini

6587191CA584E

kullanıyoruz.

 alinpikbarbell@gmail.com

OpenAI Gym araç seti ile tanışın

OpenAI Gym, RL modellerinin geliştirilmesini kolaylaştırmak için özel bir araç setidir. OpenAI Gym, önceden tanımlanmış birkaç ortamla birlikte gelir. Bazı temel örnekler CartPole ve MountainCar'dır; burada görevler, adlarından da anlaşılacağı gibi, sırasıyla bir direği dengelemek ve bir arabayı bir tepeye çıkarmaktır. Ayrıca, bir robotu bir banktaki eşyaları almak, itmek ve uzanmak için eğitmek veya blokları, topları veya kalemleri yönlendirmek için bir robot elini eğitmek için birçok gelişmiş robotik ortam da vardır. Dahası, OpenAI Gym yeni ortamlar geliştirmek için kullanışlı ve birleşik bir çerçeve sunmaktadır. Daha fazla bilgi resmi web sitesinde bulunabilir: https://gym.openai.com/.

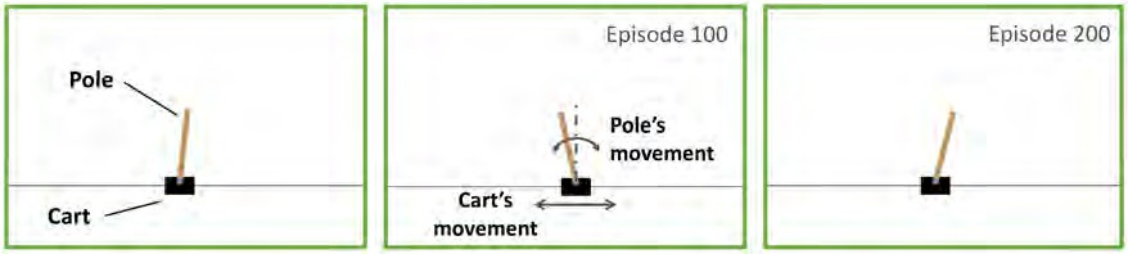
Sonraki bölümlerde OpenAI Gym kod örneklerini takip etmek için, pip kullanarak kolayca yapılabilen gym kütüphanesini (yazım sırasında 0.20.0 sürümü kullanılıyordu) yüklemeniz gerekir:

```
pip install gym==0.20
```

Kurulumla ilgili ek yardıma ihtiyacınız varsa, lütfen https:// gym.openai.com/docs/#installation adresindeki resmi kurulum kılavuzuna bakın.

OpenAI Gym'deki mevcut ortamlarla çalışma

Gym ortamları ile pratik yapmak için, OpenAI Gym'de zaten var olan CartPole-v1'den bir ortam oluşturalım. Bu örnek ortamda, *Şekil 19.6*'da gösterildiği gibi yatay olarak hareket edebilen bir arabaya bağlı bir direk vardır:



Şekil 19.6: Gym'de CartPole örneği

Direğin hareketleri fizik kanunları tarafından yönetilir ve RL aracılarının amacı direği sabitlemek ve her iki tarafa devrilmesini önlemek için arabayı nasıl hareket ettireceklerini öğrenmektir.

Şimdi, RL bağlamında CartPole ortamının durum (veya gözlem) uzayı, eylem uzayı ve bir eylemin nasıl yürütüleceği gibi bazı özelliklerine bakalım:

```
>> import gym
>> env = gym.make('CartPole-v1')
>> env.observation_space
Kutu(-3.4028234663852886e+38, 3 . 4028234663852886e+38, ( 4,), float32)
>> env.action_space
Ayrık(2)
```

Önceki kodda, CartPole problemi için bir ortam oluşturduk. Bu ortam için gözlem uzayı Box(4,) (-inf ile inf arasında kayan değerlerle) olup dört gerçek değerli sayıya karşılık gelen dört boyutlu bir uzayı temsil eder: arabanın konumu, arabanın hızı, direğin açısı ve direğin ucunun hızı. Eylem uzayı ayrık bir uzaydır, Discrete(2), iki seçenek vardır: arabayı sola ya da sağa itmek.

Daha önce gym.make('CartPole-v1') çağrısı yaparak oluşturduğumuz env ortam nesnesi, her bölümden önce bir ortamı yeniden başlatmak için kullanabileceğimiz bir reset() yöntemine sahiptir. Reset() yönteminin çağrılması temel olarak direğin başlangıç durumunu ayarlayacaktır

```
>> env.reset()
array([-0.03908273, -0.00837535, 0.03277162, -0.0207195 ])
```

(S_0):

env.reset() yöntem çağrısı tarafından döndürülen dizideki değerler, arabanın ilk konumunun -0.039, hızının -0.008 ve direğin açısının 0.033 radyan, ucunun açısal hızının ise -0.021 olduğu anlamına gelir. Reset() yöntemi çağrıldığında, bu değerler [-0.05, 0.05] aralığında tekdüze dağılıma sahip rastgele değerlerle başlatılır.

Ortamı sıfırladıktan sonra, bir eylem seçerek ve eylemi step() yöntemine geçirerek çalıştırarak ortamla etkileşime girebiliriz:

```
>> env.step(action=0)
(array([-0.03925023, -0.20395158, 0.03235723, 0.28212046]), 1.0, False, {})
>> env.step(action=1)
(array([-0.04332927, -0.00930575, 0.03799964, -0.00018409]), 1.0, False, {})
```

Önceki iki komutla, env.step(action=0) ve env.step(action=1), arabayı sırasıyla sola (action=0) ve ardından sağa (action=1) ittik. Seçilen eyleme bağlı olarak, araba ve direği fizik yasaları tarafından yönetildiği şekilde hareket edebilir. env.step() işlevini her çağırdığımızda, dört öğeden oluşan bir tuple döndürür:

- Yeni durum (veya gözlemler) için bir dizi
- Bir ödül (float türünde bir skaler değer)
- Bir sonlandırma bayrağı (Doğru veya Yanlış)
- Yardımcı bilgiler içeren bir Python sözlüğü

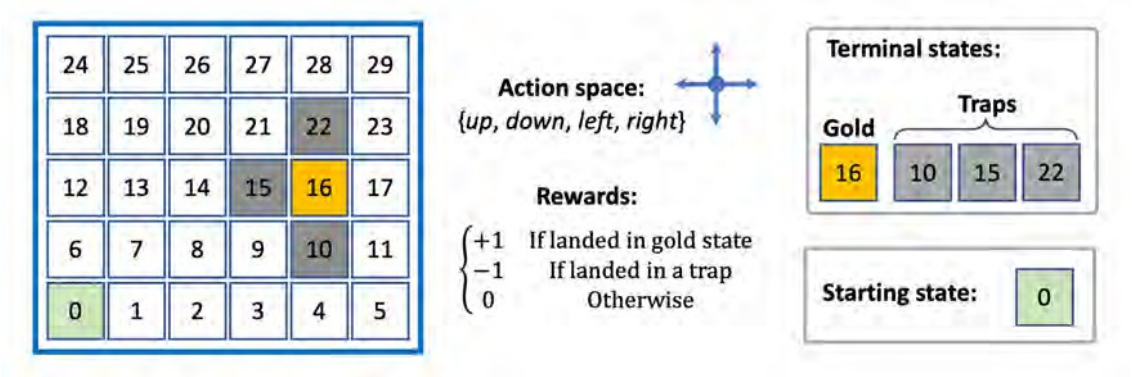
env nesnesi ayrıca her adımdan (veya bir dizi adımdan) sonra çalıştırabileceğimiz bir render() metoduna sahiptir; bu metot, ortamı ve direk ile arabanın hareketlerini zaman içinde görselleştirir.

Bölüm, direğin açısı hayali bir dikey eksene göre (her iki taraftan) 12 dereceden fazla olduğunda veya arabanın konumu merkez konumdan 2,4 birimden fazla olduğunda sona erer. Bu örnekte tanımlanan ödül, arabanın ve direğin geçerli bölgeler içinde stabilize olduğu süreyi maksimize etmektir - başka bir deyişle, toplam ödül (yani getiri) bölümün uzunluğunu maksimize ederek maksimize edilebilir.

 alinpikbarbell@gmail.com

Bir şebeke dünyası örneği

OpenAI Gym araç seti ile çalışmak için bir ısınma egzersizi olarak CartPole ortamını tanıttıktan sonra, şimdi farklı bir ortama geçeceğiz. *M* satır ve *n* sütundan oluşan basit bir ortam olan bir ızgara dünya örneği ile çalışacağız. m=5 ve n=6 olarak düşünürsek bu ortamı *Şekil 19.7'de* gösterildiği gibi özetleyebiliriz:



Şekil 19.7: Bir ızgara dünya ortamı örneği

Bu ortamda 30 farklı olası durum vardır. Bu durumlardan dördü son durumlardır: 16. durumda bir küp altın ve 10, 15 ve 22. durumlarda üç tuzak. Bu dört terminal durumdan herhangi birine iniş yapmak bölümü sona erdirir, ancak altın ve tuzak durumları arasında bir fark vardır. Altın durumuna inmek +1 gibi pozitif bir ödül getirirken, ajanı tuzak durumlarından birine taşımak -1 gibi negatif bir ödül getirir. Diğer tüm durumların ödülü 0'dır. Temsilci her zaman 0 durumundan başlar. Bu nedenle, ortamı her sıfırladığımızda, ajan 0 durumuna geri dönecektir. Eylem alanı dört yönden oluşur: yukarı, aşağı, sola ve sağa hareket.

Temsilci ızgaranın dış sınırında olduğunda, ızgarayı terk etmekle sonuçlanacak bir eylem seçmek durumu değiştirmeyecektir.

Daha sonra, OpenAI Gym paketini kullanarak bu ortamı Python'da nasıl uygulayacağımızı göreceğiz.

OpenAI Gym'de ızgara dünya ortamının uygulanması

OpenAI Gym aracılığıyla ızgara dünya ortamını denemek için, kodu etkileşimli olarak çalıştırmak yerine bir komut dosyası düzenleyicisi veya IDE kullanılması şiddetle tavsiye edilir.

İlk olarak gridworld_env.py adında yeni bir Python betiği oluşturuyoruz ve ardından gerekli paketleri ve ortamın görselleştirmesini oluşturmak için tanımladığımız iki yardımcı işlevi içe aktararak devam ediyoruz.

Ortamları görselleştirme amacıyla işlemek için OpenAI Gym kütüphanesi pyglet kütüphanesini kullanır ve bize kolaylık sağlamak için sarmalayıcı sınıflar ve işlevler sağlar. Aşağıdaki kod örneğinde ızgara dünya ortamını görselleştirmek için bu sarmalayıcı sınıfları kullanacağız. Bu sarmalayıcı sınıflar hakkında daha fazla ayrıntı https://github.com/openai/gym/blob/58ed658d9b15fd410c50d1fdb25a7cad9acb7fa4/gym/envs/classic_control/rendering.py adresinde bulunabilir.

Aşağıdaki kod örneği bu sarmalayıcı sınıfları kullanmaktadır:

```
## B e t i k : gridworld_env.py
import numpy as np
from gym.envs.toy_text import discrete
from collections import defaultdict import
time
import pickle
import os
from gym.envs.classic_control import rendering

CELL_SIZE = 100
MARJİN = 10


def get_coords(row, col, loc='center'): xc
    = (col+1.5) * CELL_SIZE
    yc = (row+1.5) * CELL_SIZE
    if loc == 'center':
        xc, yc döndür
    elif loc == 'interior_corners': half_size
        = CELL_SIZE//2 - MARGIN
        xl, xr = xc - half_size, xc + half_size yt,
        yb = xc - half_size, xc + half_size
        return [(xl, yt), (xr, yt), (xr, yb), (xl, yb)]
    elif loc == 'interior_triangle':
        x1, y1 = xc, yc + CELL_SIZE//3
        x2, y2 = xc + CELL_SIZE//3, yc - CELL_SIZE//3
        x3, y3 = xc - CELL_SIZE//3, yc - CELL_SIZE//3
        return [(x1, y1), (x2, y2), (x3, y 3 ) ]

def draw_object(coords_list):
    if len(coords_list) == 1: # -> daire
        obj = rendering.make_circle(int(0.45*CELL_SIZE))
        obj_transform = rendering.Transform()
        obj.add_attr(obj_transform)
        obj_transform.set_translation(*coords_list[0])
        obj.set_color(0.2, 0.2, 0 . 2) # -> siyah
    elif len(coords_list) == 3: # -> üçgen
        obj = rendering.FilledPolygon(coords_list)
        obj.set_color(0.9, 0.6, 0 . 2) # -> sarı
    elif len(coords_list) > 3: # -> çokgen
        obj = rendering.FilledPolygon(coords_list)
```

 alinpikbarbell@gmail.com

```
        obj.set_color(0.4, 0.4, 0.8) # -> mavi
    return obj
```

Gym 0.22 veya daha yenisini kullanma

Spor salonunun şu anda bazı dahili yeniden yapılandırmalardan geçtiğini unutmayın. 0.22 ve daha yeni sürümlerde, önceki kod örneğini (gridworld_env.py'den) güncellemeniz ve aşağıdaki satırı değiştirmeniz gerekebilir

```
from gym.envs.classic_control import rendering
```

aşağıdaki kod ile:

```
from gym.utils import pyglet_rendering
```

Daha fazla ayrıntı için lütfen https://github.com/rasbt/ adresindeki kod deposuna bakın
machine-learning-book/tree/main/ch19

İlk yardımcı fonksiyon olan get_coords(), ızgara dünya ortamına açıklama eklemek için kullanacağımız geometrik şekillerin koordinatlarını döndürür; örneğin altını göstermek için bir üçgen veya tuzakları göstermek için daireler gibi. Koordinat listesi draw_object() fonksiyonuna aktarılır ve bu fonksiyon koordinat listesinin uzunluğuna bağlı olarak bir daire, üçgen ya da çokgen çizmeye karar verir.

Şimdi, grid dünyası ortamını tanımlayabiliriz. Aynı dosyada (gridworld_env.py), OpenAI Gym'in DiscreteEnv sınıfından devralınan GridWorldEnv adlı bir sınıf tanımlıyoruz. Bu sınıfın en önemli işlevi, eylem alanını tanımladığımız, her bir eylemin rolünü belirttiğimiz ve terminal durumlarını (altın ve tuzaklar) aşağıdaki gibi belirlediğimiz **init () adlı** kurucu yöntemidir:

```
sınıf GridWorldEnv(discrete.DiscreteEnv):
    def init (self, num_rows=4, num_cols=6, delay=0.05): self.num_rows
        = num_rows
        self.num_cols = num_cols
        self.delay = delay
        move_up = lambda row, col: (max(row-1, 0), col)
        move_down = lambda row, col: (min(row+1, num_rows-1), col)
        move_left = lambda row, col: (row, max(col-1, 0))
        move_right = lambda row, col: (
            row, min(col+1, num_cols-1))
        self.action_defs={0: move_up, 1: move_right,
                          2: move_down, 3: move_left}
        ## Eyalet/eylem sayısı
        nS = num_cols*num_rows
        nA = len(self.action_defs)
        self.grid2state_dict={(s//num_cols, s%num_cols):s
                              for s in range(nS)}
        self.state2grid_dict={s:(s//num_cols, s%num_cols)
```

```
                                for s in range(nS)}
        ## Altın devlet
        gold_cell = (num_rows//2, num_cols-2)

        ## Tuzak durumları
        tuzak_hücreleri = [((altın_hücre[0]+1), altın_hücre[1]),
                           (gold_cell[0], gold_cell[1]-1),
                           ((gold_cell[0]-1), gold_cell[1])]
        gold_state = self.grid2state_dict[gold_cell]
        trap_states = [self.grid2state_dict[(r, c)]
                           for (r, c) in trap_cells]
        self.terminal_states = [gold_state] + trap_states
        print(self.terminal_states)
        ## Geçiş olasılığını oluşturun
        P = defaultdict(dict)
        for s in range(nS):
            row, col = self.state2grid_dict[s]
            P[s] = defaultdict(liste)
            a in range(nA) için:
                action = self.action_defs[a]
                next_s = self.grid2state_dict[action(row, col)]

                ## Terminal durumu
                if self.is_terminal(next_s):
                    r = (1.0 if next_s == self.terminal_states[0]
                        else -1.0)
                Başka:
                    r = 0.0
                if self.is_terminal(s):
                    done = True
                    next_s = s
                else:
                    done = False
                P[s][a] = [(1.0, next_s, r, done)]
        ## İlk durum dağılımı
        isd = np.zeros(nS)
        isd[0] = 1.0
        super(). init (nS, nA, P, isd) self.viewer =
        None self._build_display(gold_cell,
        trap_cells)
```

 alinpikbarbell@gmail.com

```
def is_terminal(self, state):
    return state in self.terminal_states


def _build_display(self, gold_cell, trap_cells): screen_width
    = (self.num_cols+2) * CELL_SIZE screen_height =
    (self.num_rows+2) * CELL_SIZE self.viewer =
    rendering.Viewer(screen_width,
                                        screen_height)
    all_objects = []
    ## Sınır noktalarının koordinatlarının listesi
    bp_list = [
        (CELL_SIZE-MARGIN, CELL_SIZE-MARGIN), (screen_width-
        CELL_SIZE+MARGIN, CELL_SIZE-MARGIN),
        (screen_width-CELL_SIZE+MARGIN,
         screen_height-CELL_SIZE+MARGIN),
        (CELL_SIZE-MARGIN, screen_height-CELL_SIZE+MARGIN)
    ]
    border = rendering.PolyLine(bp_list, True)
    border.set_linewidth(5)
    all_objects.append(border)
    ## Dikey çizgiler
    for col in range(self.num_cols+1):
        x1, y1 = (col+1)*CELL_SIZE, CELL_SIZE
        x2, y2 = (col+1)*CELL_SIZE,\
                 (self.num_rows+1)*CELL_SIZE
        line = rendering.PolyLine([(x1, y1), (x2, y2)], False) all_objects.append(line)

    ## Yatay çizgiler
    for row in range(self.num_rows+1):
        x1, y1 = CELL_SIZE, (row+1)*CELL_SIZE
        x2, y2 = (self.num_cols+1)*CELL_SIZE,\
                 (row+1)*CELL_SIZE
        line=rendering.PolyLine([(x1, y1), (x2, y2)], False)
        all_objects.append(line)

    ## Tuzaklar: --> daireler
    for cell in trap_cells:
        trap_coords = get_coords(*cell, loc='center')
        all_objects.append(draw_object([trap_coords]))
```

```
        ## Altın: --> üçgen
        gold_coords = get_coords(*gold_cell,
                                 loc='interior_triangle')
        all_objects.append(draw_object(gold_coords))
        ## Ajan --> kare veya robot
        if (os.path.exists('robot-coordinates.pkl') and
                CELL_SIZE==100):
            agent_coords = pickle.load(
                open('robot-coordinates.pkl', 'rb'))
            starting_coords = get_coords(0, 0, loc='center')
            agent_coords += np.array(starting_coords)
        Başka:
            agent_coords = get_coords(
                0, 0, loc='interior_corners') agent
        = draw_object(agent_coords)
        self.agent_trans = rendering.Transform()
        agent.add_attr(self.agent_trans)
        all_objects.append(agent)
        for obj in all_objects:
            self.viewer.add_geom(obj)


    def render(self, mode='human', done=False):
        if done:
            sleep_time = 1
        else:
            sleep_time = self.delay
        x_coord = self.s %
        self.num_cols y_coord = self.s
        // self.num_cols x_coord =
        (x_coord+0) * CELL_SIZE y_coord =
        (y_coord+0) * CELL_SIZE
        self.agent_trans.set_translation(x_coord, y_coord)
        rend = self.viewer.render(
            return_rgb_array=(mode=='rgb_array'))
        time.sleep(sleep_time)
        dönüş rend


    def close(self):
        if self.viewer:
            self.viewer.close()
            self.viewer = None
```

 olimpikbarbell@gmail.com

Bu kod, bu ortamın örneklerini oluşturabileceğimiz ızgara dünya ortamını uygular. Daha sonra CartPole örneğindekine benzer bir şekilde onunla etkileşime geçebiliriz. Uygulanan sınıf, GridWorldEnv, durumu sıfırlamak için reset() ve bir eylemi yürütmek için step() gibi yöntemleri miras alır. Uygulamanın ayrıntıları aşağıdaki gibidir:

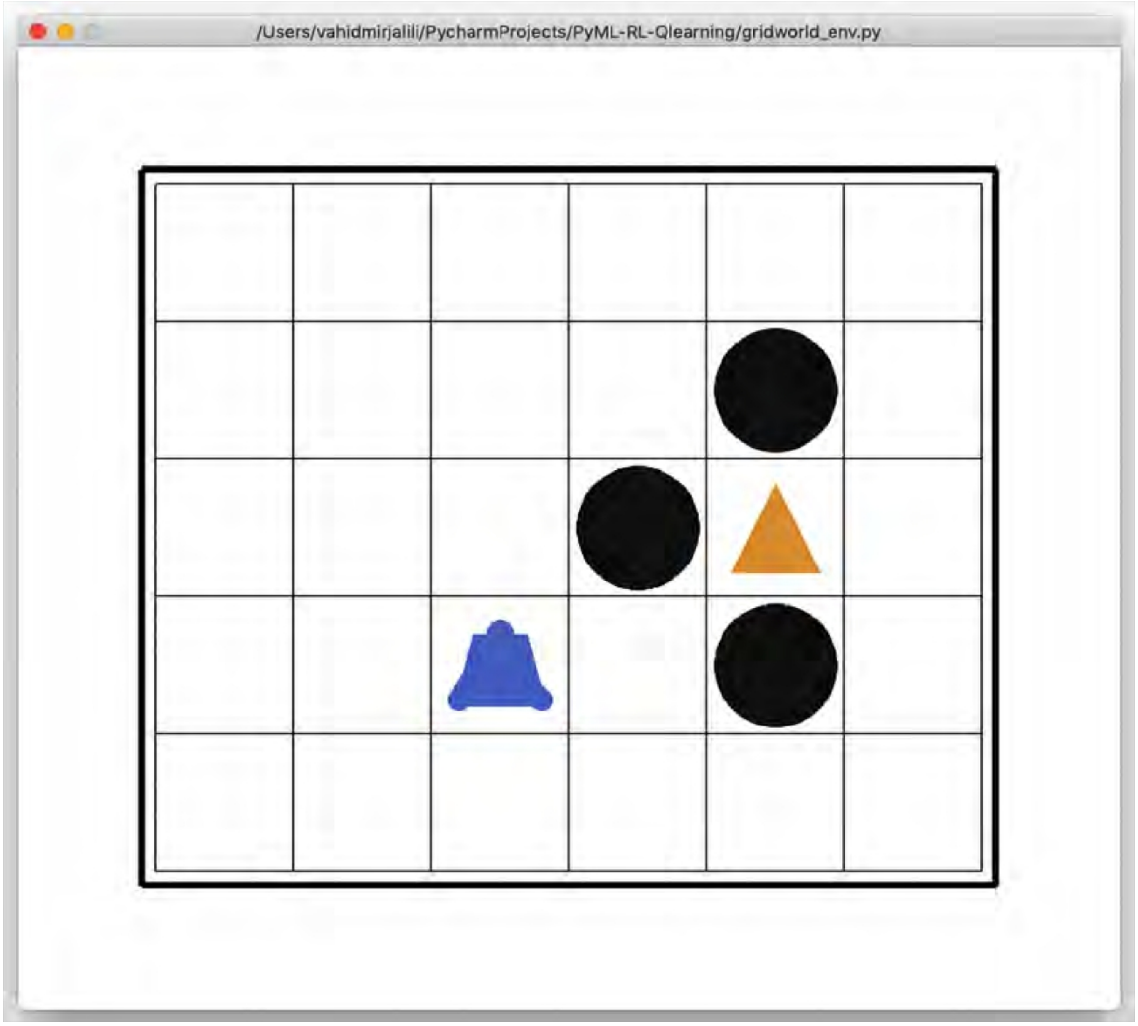
- Dört farklı eylemi lambda fonksiyonları kullanarak tanımladık: move_up(), move_down(), move_left() ve move_right().
- NumPy dizisi isd başlangıç durumlarının olasılıklarını tutar, böylece reset() yöntemi (ana sınıftan) çağrıldığında bu dağılıma göre rastgele bir durum seçilecektir. Her zaman 0 durumundan (ızgara dünyasının sol alt köşesi) başladığımız için, 0 durumunun olasılığını 1,0 ve diğer 29 durumun tümünün olasılıklarını 0,0 olarak ayarlıyoruz.
- Python sözlüğü P'de tanımlanan geçiş olasılıkları, bir eylem seçildiğinde bir durumdan başka bir duruma geçme olasılıklarını belirler. Bu, bir eylemde bulunmanın ortamın stokastikliğine bağlı olarak farklı sonuçlar doğurabileceği olasılıksal bir ortama sahip olmamızı sağlar. Basitlik açısından, durumu seçilen eylem yönünde değiştirecek tek bir sonuç kullanacağız. Son olarak, bu geçiş olasılıkları env.step() fonksiyonu tarafından bir sonraki durumu belirlemek için kullanılacaktır.
- Ayrıca, _build_display() işlevi ortamın ilk görselleştirmesini ayarlayacak ve render() işlevi aracının hareketlerini gösterecektir.

Öğrenme sürecinde geçiş olasılıklarını bilmediğimizi ve amacımızın çevreyle etkileşime girerek öğrenmek olduğunu unutmayın. Bu nedenle, sınıf tanımı dışında P'ye erişimimiz yoktur.

Şimdi, yeni bir ortam oluşturarak ve her durumda rastgele eylemler gerçekleştirerek rastgele bir episode görselleştirerek bu uygulamayı test edebiliriz. Aşağıdaki kodu aynı Python betiğinin (gridworld_env.py) sonuna ekleyin ve ardından betiği çalıştırın:

```
if name == ' main ': env =
    GridWorldEnv(5, 6) for i
    in range(1):
        s = env.reset()
        env.render(mode='human', done=False)
        while True:
            eylem = np.random.choice(env.nA)
            res = env.step(eylem)
            print('Eylem ', env.s, eylem, ' -> ', res)
            env.render(mode='human', done=res[2])
            if res[2]:
                Mola
    env.close()
```

Komut dosyasını çalıştırdıktan sonra, *Şekil 19.8*'de gösterildiği gibi ızgara dünyası ortamının bir görselleştirmesini görmelisiniz:



Şekil 19.8: Izgara dünyası ortamımızın görselleştirilmesi

Q-öğrenme ile ızgara dünya problemini çözme

RL algoritmalarının teorisine ve geliştirme sürecine odaklandıktan ve OpenAI Gym araç seti aracılığıyla ortamı kurduktan sonra, şimdi şu anda en popüler RL algoritması olan Q-öğrenmeyi uygulayacağız. Bunun için, gridworld_env.py betiğinde zaten uyguladığımız ızgara dünyası örneğini kullanacağız.

 alinpikbarbell@gmail.com

Şimdi, yeni bir betik oluşturuyoruz ve buna agent.py adını veriyoruz. Bu agent.py betiğinde, çevre ile etkileşim için aşağıdaki gibi bir ajan tanımlıyoruz:

```
## B e t i k : agent.py
from collections import defaultdict
import numpy as np

sınıf ajanı:
    def init (
            self, env,
            learning_rate=0.01,
            discount_factor=0.9,
            epsilon_greedy=0.9,
            epsilon_min=0.1,
            epsilon_decay=0.95):
        self.env = env
        self.lr = learning_rate
        self.gamma = discount_factor
        self.epsilon = epsilon_greedy
        self.epsilon_min = epsilon_min
        self.epsilon_decay = epsilon_decay
        ## q_tablosunu tanımlayın
        self.q_table = defaultdict(lambda: np.zeros(self.env.nA))

    def choose_action(self, state):
        if np.random.uniform() < self.epsilon: action
            = np.random.choice(self.env.nA)
        Başka:
            q_vals = self.q_table[state]
            perm_actions = np.random.permutation(self.env.nA)
            q_vals = [q_vals[a] for a in perm_actions]
            perm_q_argmax = np.argmax(q_vals)
            action = perm_actions[perm_q_argmax]
        return action

    def _learn(self, transition):
        s, a, r, next_s, done = geçiş q_val =
        self.q_table[s][a]
        eğer bittiyse:
            q_hedef = r
```

```
        Başka:
            q_hedef = r + self.gamma*np.max(self.q_table[next_s])
        ## q_tablosunu güncelleyin
        self.q_table[s][a] += self.lr * (q_target - q_val)
        ## Epsilonu ayarlayın
        self._adjust_epsilon()

    def _adjust_epsilon(self):
        if self.epsilon > s e l f . epsilon_min:
            self.epsilon *= self.epsilon_decay
```

init () yapıcısı öğrenme oranı, iskonto faktörü ($\gamma\gamma$) ve $\epsilon\epsilon$-greedy politikası için parametreler gibi çeşitli hiperparametreleri ayarlar. Başlangıçta, yüksek bir $\epsilon\epsilon$ değeri ile başlıyoruz, ancak _adjust_epsilon() yöntemi minimum değere, $\epsilon\epsilon_{\text{min}}$ ulaşana kadar azaltır. choose_action() yöntemi, $\epsilon\epsilon$-greedy politikasına dayalı olarak aşağıdaki şekilde bir eylem seçer. Eylem-değer fonksiyonuna dayalı olarak eylemin rastgele mi yoksa başka bir şekilde mi seçilmesi gerektiğini belirlemek için rastgele bir tekdüze sayı seçilir. _learn() yöntemi, Q-öğrenme algoritması için güncelleme kuralını uygular. Her geçiş için mevcut durum (s), seçilen eylem (a), gözlemlenen ödül (r), bir sonraki durum (s') ve bölümün sonuna ulaşılıp ulaşılmadığını belirlemek için bir bayraktan oluşan bir tuple alır. Hedef değer, bölüm sonu olarak işaretlenmişse gözlemlenen ödüle (r) eşittir; Aksi takdirde, hedef $r + rr\, rr\, \underset{a}{QQ}(s', a)$ olur.

Son olarak, bir sonraki adımımız için, her şeyi bir araya getirmek ve Q-öğrenme algoritmasını kullanarak aracıyı eğitmek için qlearning.py adlı yeni bir komut dosyası oluşturuyoruz.

Aşağıdaki kodda, ajanın _choose_action() yöntemini çağırarak ve ortamı çalıştırarak bir bölümü simüle eden Q-öğrenme algoritmasını uygulayan run_qlearning() adlı bir fonksiyon tanımlıyoruz. Ardından, eylem-değer fonksiyonunu güncellemek için geçiş tuple'ı ajanın _learn() metoduna aktarılır. Buna ek olarak, öğrenme sürecini izlemek için, her bölümün nihai ödülünü (-1 veya +1 olabilir) ve bölümlerin uzunluğunu (bölümün başlangıcından sonuna kadar ajan tarafından alınan hamle sayısı) da saklarız.

Ödüllerin listesi ve hamle sayısı daha sonra plot_learning_history() kullanılarak çizilir. fonksiyon:

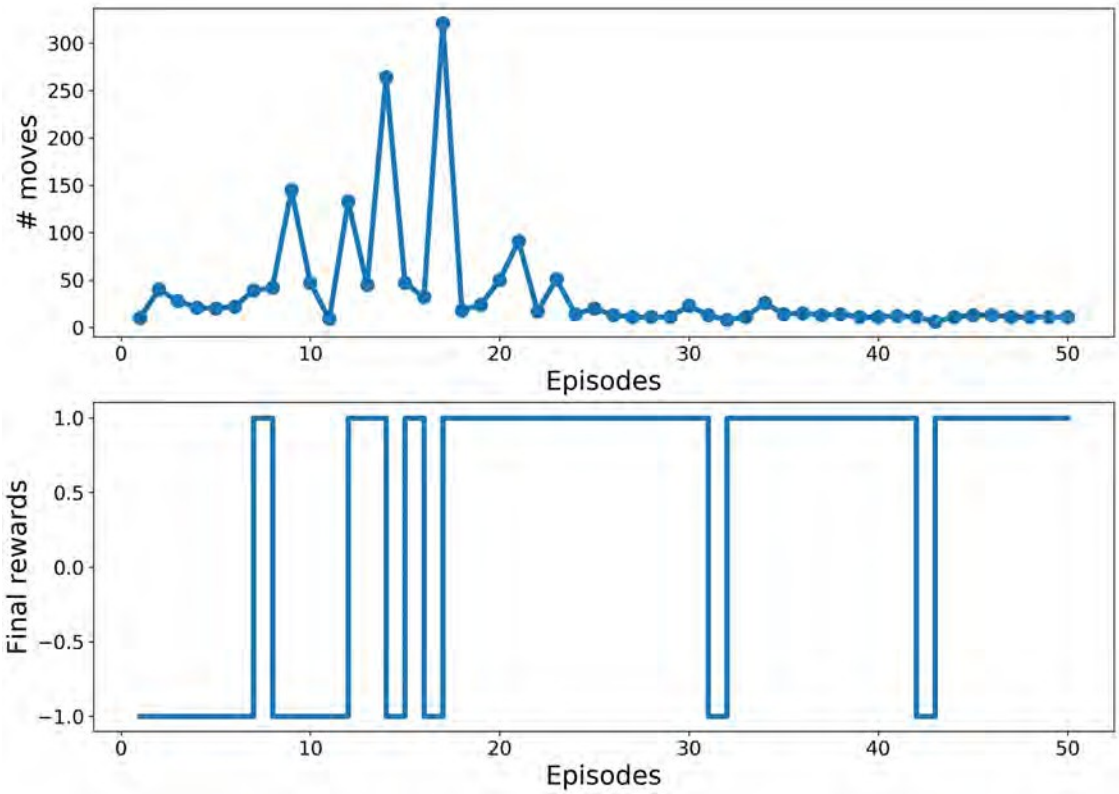
```
## B e t i k : qlearning.py
from gridworld_env import GridWorldEnv
from agent import Agent
from collections import namedtuple
import matplotlib.pyplot as plt import
numpy as np np.random.seed(1)
```

 olimpikbarbell@gmail.com

```
Geçiş = namedtuple(
    'Geçiş', ('durum', 'eylem', 'ödül',
                        'next_state', 'done'))


def run_qlearning(agent, env, num_episodes=50):
    history = []
    for episode in range(num_episodes):
        state = env.reset()
        env.render(mode='human')
        final_reward, n_moves = 0.0, 0
        while True:
            action = agent.choose_action(state) next_s,
            reward, done, _ = env.step(action)
            agent._learn(Transition(durum, eylem, ödül,
                                    next_s, d o n e ))
            env.render(mode='human', done=done)
            state = next_s
            n_moves += 1
            if done:
                break final_reward
            = ödül
        history.append((n_moves, final_reward)) print(f'Bölüm
        {episode}: Ödül {final_reward:.2} '
                f'#Moves {n_moves}')
    return history


def plot_learning_history(history):
    fig = plt.figure(1, figsize=(14, 10))
    ax = fig.add_subplot(2, 1, 1)
    episodes = np.arange(len(history))
    moves = np.array([h[0] for h in history]) plt.plot(episodes,
    moves, lw=4,
                marker='o', markersize=10)
    ax.tick_params(axis='both', which='major', labelsize=15)
    plt.xlabel('Episodes', size=20)
    plt.ylabel('# moves', size=20)
    ax = fig.add_subplot(2, 1, 2)
    ödüller = np.array([h[1] for h in history])
```

```
    plt.step(episodes, rewards, lw=4)
    ax.tick_params(axis='both', which='major', labelsize=15)
    plt.xlabel('Episodes', size=20)
    plt.ylabel('Final rewards', size=20)
    plt.savefig('q-learning-history.png', dpi=300)
    plt.show()

if name == '__main__':
    env = GridWorldEnv(num_rows=5, num_cols=6)
    agent = Agent(env)
    geçmiş = run_qlearning(ajan, env)
    env.close()
    plot_learning_history(geçmiş)
```

Bu komut dosyası çalıştırıldığında Q-öğrenme programı 50 bölüm boyunca çalıştırılacaktır. Ajanın davranışı görselleştirilecek ve öğrenme sürecinin başında ajanın çoğunlukla tuzak durumlarına düştüğünü görebilirsiniz. Ancak zamanla, başarısızlıklarından öğrenir ve sonunda altın durumu bulur (örneğin, 7. bölümde ilk kez). *Şekil 19.9* ajanın hamle sayısını ve ödüllerini göstermektedir:



Şekil 19.9: Temsilcinin hamle sayısı ve ödülleri

alinpikbarbell@gmail.com

Önceki şekilde gösterilen öğrenme geçmişi, ajanın 30 bölümden sonra altın duruma ulaşmak için kısa bir yol öğrendiğini göstermektedir. Sonuç olarak, 30. bölümden sonraki bölümlerin uzunlukları, $\epsilon\epsilon$-greedy politikası nedeniyle küçük sapmalarla aşağı yukarı aynıdır.

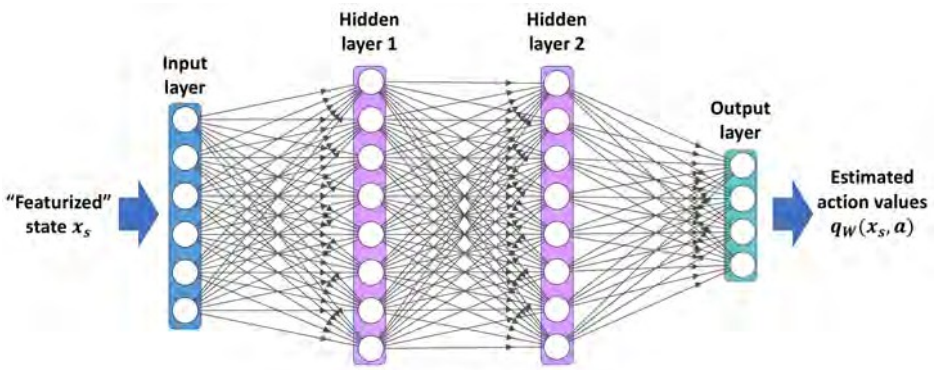
Derin Q-öğrenmeye bir bakış

Önceki kodda, ızgara dünya örneği için popüler Q-öğrenme algoritmasının bir uygulamasını gördük. Bu örnek, Q değerlerini bir Python sözlüğünde saklamanın yeterli olduğu 30 boyutunda ayrık bir durum uzayından oluşuyordu.

Bununla birlikte, bazen durum sayısının çok büyük, muhtemelen neredeyse sonlu büyüklükte olabileceğine dikkat etmeliyiz. Ayrıca, ayrık durumlarla çalışmak yerine sürekli bir durum uzayı ile uğraşıyor olabiliriz. Dahası, bazı durumlar eğitim sırasında hiç ziyaret edilmeyebilir, bu da ajanı daha sonra bu tür görülmeyen durumlarla başa çıkacak şekilde genelleştirirken sorun yaratabilir.

Bu sorunları ele almak için, eylem-değer fonksiyonu için değer fonksiyonunu $V(S_t)$ veya $Q(S_t, A_t)$ gibi bir tablo biçiminde temsil etmek yerine, bir *fonksiyon yaklaşımı* yaklaşımı kullanıyoruz. Burada, gerçek değer fonksiyonuna, yani $v_{ww}(x_s) \approx v_\pi(s)$, yaklaşmayı öğrenebilen parametrik bir fonksiyon, $v(x_{ws})$, tanımlıyoruz; burada x_s bir dizi girdi özelliğidir (veya "özellikli" durumlardır).

Yaklaşım fonksiyonu, $q_w(x_s, a)$, bir **derin sinir ağı** (**DNN**) olduğunda, ortaya çıkan model **derin Q-ağ** (**DQN**) olarak adlandırılır. Bir DQN modelini eğitmek için, ağırlıklar Q-öğrenme algoritmasına göre güncellenir. DQN modelinin bir örneği *Şekil 19.10'*da gösterilmektedir; burada durumlar ilk katmana aktarılan özellikler olarak temsil edilmektedir:



Şekil 19.10: Bir DQN örneği

Şimdi, *derin Q-öğrenme* algoritmasını kullanarak bir DQN'yi nasıl eğitebileceğimizi görelim. Genel olarak, ana yaklaşım tablo Q-öğrenme yöntemine çok benzer. Temel fark, artık eylem değerlerini hesaplayan çok katmanlı bir NN'ye sahip olmamızdır.

Q-öğrenme algoritmasına göre bir DQN modelinin eğitilmesi

Bu bölümde, Q-öğrenme algoritmasını kullanarak bir DQN modelini eğitme prosedürünü açıklıyoruz. Derin Q-öğrenme yaklaşımı, daha önce uyguladığımız standart Q-öğrenme yaklaşımında bazı değişiklikler yapmamızı gerektirmektedir.

Bu değişikliklerden biri, Q-öğrenme için önceki bölümün kodunda basitçe bir sözlükte saklanan eylem değerlerine erişen ajanın choose_action() yöntemindedir. Şimdi, bu işlev, eylem değerlerini hesaplamak için NN modelinin ileri geçişini gerçekleştirecek şekilde değiştirilmelidir.

Derin Q-öğrenme algoritması için gereken diğer değişiklikler aşağıdaki iki alt bölümde açıklanmaktadır.

Tekrar oynatma belleği

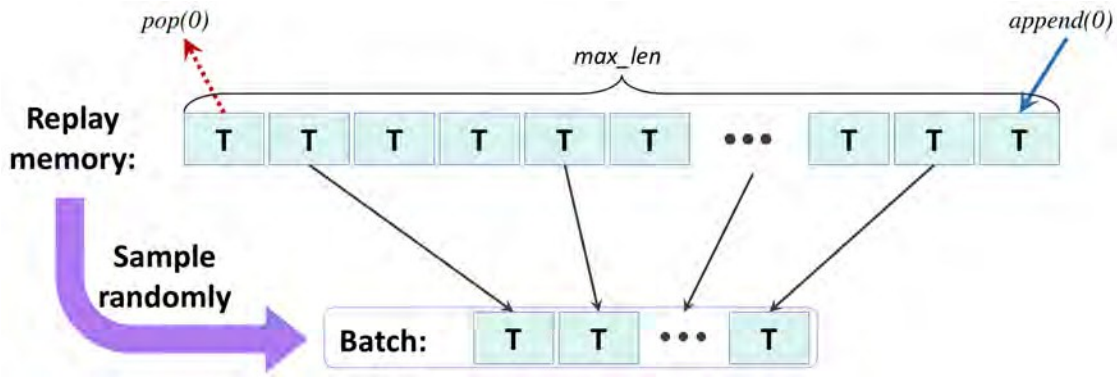
Q-öğrenme için önceki tablo yöntemini kullanarak, belirli durum-eylem çiftlerinin değerlerini diğerlerinin değerlerini etkilemeden güncelleyebiliyorduk. Ancak şimdi $q(s, a)$ değerini bir NN modeliyle yaklaşık olarak hesapladığımıza göre, bir durum-eylem çiftinin ağırlıklarının güncellenmesi muhtemelen diğer durumların çıktılarını da etkileyecektir. Denetimli bir görev (örneğin bir sınıflandırma görevi) için stokastik gradyan inişi kullanarak NN'leri eğitirken, yakınsayana kadar eğitim verilerini birçok kez yinelemek için birden fazla epok kullanırız.

Eğitim sırasında bölümler değişeceğinden ve sonuç olarak eğitimin ilk aşamalarında ziyaret edilen bazı durumların daha sonra ziyaret edilme olasılığı azalacağından, Q-öğrenmede bu mümkün değildir.

Ayrıca, bir başka sorun da bir NN'yi eğitirken eğitim örneklerinin **IID** (**bağımsız ve özdeş dağılımlı**) olduğunu varsaymamızdır. Ancak, ajanın bir bölümünden alınan örnekler, bir dizi geçiş oluşturdukları için IID değildir.

Bu sorunları çözmek için, ajan çevreyle etkileşime girdikçe ve bir geçiş beşlisi $q_w(x_s, a)$ ürettikçe, bu tür geçişlerin büyük (ancak sonlu) bir sayısını, genellikle *yeniden oynatma belleği* olarak adlandırılan bir bellek tamponunda saklarız. Her yeni etkileşimden sonra (yani, ajan bir eylem seçer ve bunu ortamda gerçekleştirir), ortaya çıkan yeni geçiş beşlisi belleğe eklenir.

Belleğin boyutunu sınırlı tutmak için, en eski geçiş bellekten kaldırılacaktır (örneğin, bir Python listesi ise, listenin ilk öğesini kaldırmak için pop(0) yöntemini kullanabiliriz). Ardından, kaybı hesaplamak ve ağ parametrelerini güncellemek için kullanılacak olan mini bir örnek grubu bellek tamponundan rastgele seçilir. *Şekil 19.11* süreci göstermektedir:



Şekil 19.11: Yeniden oynatma belleği süreci

 alinpikbarbell@gmail.com

Yeniden oynatma belleğinin uygulanması

Yeniden oynatma belleği, listeye her yeni öğe eklediğimizde listenin boyutunu kontrol etmemiz ve gerekirse pop(0)'ı çağırmamız gereken bir Python listesi kullanılarak uygulanabilir.

Alternatif olarak, Python koleksiyon kütüphanesindeki deque veri yapısını kullanabiliriz, bu da isteğe bağlı bir argüman olan max_len'i belirtmemize izin verir. max_len argümanını belirterek, sınırlı bir deque'ye sahip olacağız. Bu nedenle, nesne dolduğunda, yeni bir eleman eklemek, otomatik olarak bir elemanın çıkarılmasıyla sonuçlanır.

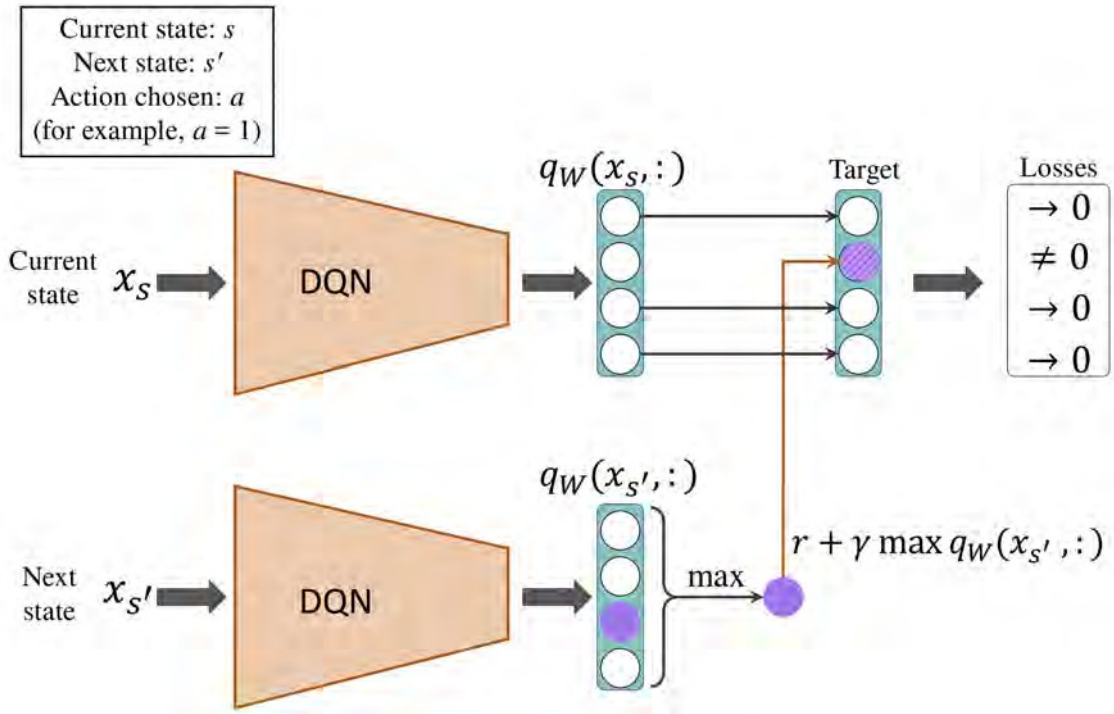
Bunun bir Python listesi kullanmaktan daha verimli olduğunu unutmayın, çünkü pop(0) kullanarak bir listenin ilk öğesini kaldırmak O(n) karmaşıklığına sahipken, deque'nin çalışma zamanı karmaşıklığı O(1)'dir. https://docs.python.org/3.9/library/collections. html#collections.deque adresinde bulunan resmi dokümandan deque uygulaması hakkında daha fazla bilgi edinebilirsiniz.

Kaybı hesaplamak için hedef değerlerin belirlenmesi

Tablo Q-öğrenme yönteminde yapılması gereken bir diğer değişiklik de DQN model parametrelerinin eğitimi için güncelleme kuralının nasıl uyarlanacağıdır. Örnekler yığınında saklanan bir geçiş beşlisinin, T, $(x_s, a, r, x_{sF}, \text{done})$ içerdiğini hatırlayın.

*Şekil 19.12'*de gösterildiği gibi, DQN modelinin iki ileri geçişini gerçekleştiriyoruz. İlk ileri geçiş mevcut durumun özelliklerini kullanır (x_s). Ardından, ikinci ileri geçiş bir sonraki durumun (x_{sF}) özelliklerini kullanır. Sonuç olarak, sırasıyla birinci ve ikinci ileri geçişten tahmini eylem değerlerini, $q_{ww}(x_s, :)$ ve $q_{ww}(x_{sF}, :)$, elde edeceğiz. (Burada, bu $q_{ww}(x_s, :)$ notasyonu, tüm Q değerleri için bir vektör anlamına gelir.
$\hat{A}$'daki eylemler). Geçiş beşlisinden, a eyleminin temsilci tarafından seçildiğini biliyoruz.

Bu nedenle, Q-öğrenme algoritmasına göre, durum-eylem çiftine (x_s, a) karşılık gelen eylem değerini skaler hedef değer $r + \gamma \max_{a' \in A} q_{ww}(x_{sF}, a')$ ile güncellememiz gerekir. Bunun yerine bir skaler hedef değeri, *Şekil 19.12'*de gösterildiği gibi, diğer eylemler için eylem değerlerini koruyan bir hedef eylem değeri vektörü oluşturacağız, $a' \neq a$:



Şekil 19.12: DQN kullanarak hedef değeri belirleme

Bunu, aşağıdaki üç büyüklüğü kullanarak bir regresyon problemi olarak ele alıyoruz:

- Şu anda tahmin edilen değerler, q_{ww} (x_s ,:)
- Açıklandığı gibi hedef değer vektörü
- Standart **ortalama karesel hata** (**MSE**) kayıp fonksiyonu

Sonuç olarak, kayıplar *a* hariç her eylem için sıfır olacaktır. Son olarak, hesaplanan kayıp, ağ parametrelerini güncellemek için geri yayılacaktır.

 alinpikbarbell@gmail.com

Derin bir Q-öğrenme algoritmasının uygulanması

Son olarak, tüm bu teknikleri derin bir Q-öğrenme algoritması uygulamak için kullanacağız. Bu kez, daha önce tanıttığımız OpenAI Gym ortamındaki CartPole ortamını kullanacağız. CartPole ortamının 4 boyutunda sürekli bir durum uzayına sahip olduğunu hatırlayın. Aşağıdaki kodda, modeli oluşturan ve çeşitli hiperparametreleri belirleyen bir sınıf olan DQNAgent'ı tanımlıyoruz.

Bu sınıf, tablo Q-öğrenmeye dayanan önceki aracıya kıyasla iki ek yönteme sahiptir. remember() metodu bellek tamponuna yeni bir geçiş beşlisi ekler ve replay() metodu örnek geçişlerin mini bir grubunu oluşturur ve bunu ağın ağırlık parametrelerini güncellemek için _learn() metoduna aktarır:

```
spor salonu İthal edin
import numpy as np
import torch
import torch.nn as nn import
random
import matplotlib.pyplot as plt from
collections import namedtuple from
collections import deque
np.random.seed(1)
torch.manual_seed(1)

Geçiş = namedtuple(
                'Geçiş', ('durum', 'eylem', 'ödül',
                                'next_state', 'done'))
class DQNAgent:
    def init (
                self, env, discount_factor=0.95,
                epsilon_greedy=1.0, epsilon_min=0.01,
                epsilon_decay=0.995, learning_rate=1e-3,
                max_memory_size=2000):
            self.env = env
            self.state_size = e n v . observation_space.shape[0]
            s e l f . action_size = env.action_space.n
            s e l f . memory =
            deque(maxlen=max_memory_size) self.gamma =
            discount_factor
            self.epsilon = epsilon_greedy
            self.epsilon_min = epsilon_min
            self.epsilon_decay = epsilon_decay
            self.lr = learning_rate
```

```
        self._build_nn_model()

    def _build_nn_model(self):
        self.model = n n . Sequential(nn.Linear(self.state_size, 256),
                                      nn.ReLU(),
                                      nn.Linear(256, 128),
                                      nn.ReLU(),
                                      nn.Linear(128, 64),
                                      nn.ReLU(),
                                      nn.Linear(64, self.action_size))
        self.loss_fn = nn.MSELoss()
        self.optimizer = torch.optim.Adam(
                                        self.model.parameters(), self.lr)

    def hatırla(self, geçiş):
        self.memory.append(geçiş)

    def choose_action(self, state):
        if np.random.rand() <= self.epsilon:
            return np.random.choice(self.action_size)
        with torch.no_grad():
            q_values = self.model(torch.tensor(state,
                                    dtype=torch.float32))[0]
        return torch.argmax(q_values).item() # eylem döndürür

    def _learn(self, batch_samples):
        batch_states, batch_targets = [], []
        for transition in batch_samples:
            s, a, r, next_s, done =
            torch.no_grad() ile geçiş:
                eğer bittiyse:
                    hedef = r
                else:
                    pred = self.model(torch.tensor(next_s,
                                        dtype=torch.float32))[0]
                    hedef = r + self.gamma * pred.max()
            target_all = self.model(torch.tensor(s,
                                        dtype=torch.float32))[0]
            target_all[a] = hedef
```

 olimpikbarbell@gmail.com

```
                batch_states.append(s.flatten())
                batch_targets.append(target_all)
                self._adjust_epsilon()
                self.optimizer.zero_grad()
                pred = self.model(torch.tensor(batch_states,
                                  dtype=torch.float32))
                loss = self.loss_fn(pred, torch.stack(batch_targets))
                loss.backward()
                self.optimizer.step()
            return loss.item()

        def _adjust_epsilon(self):
            if self.epsilon > s e l f . epsilon_min:
                self.epsilon *= self.epsilon_decay

        def replay(self, batch_size):
            samples = random.sample(self.memory, batch_size) return
            s e l f . _learn(samples)
```

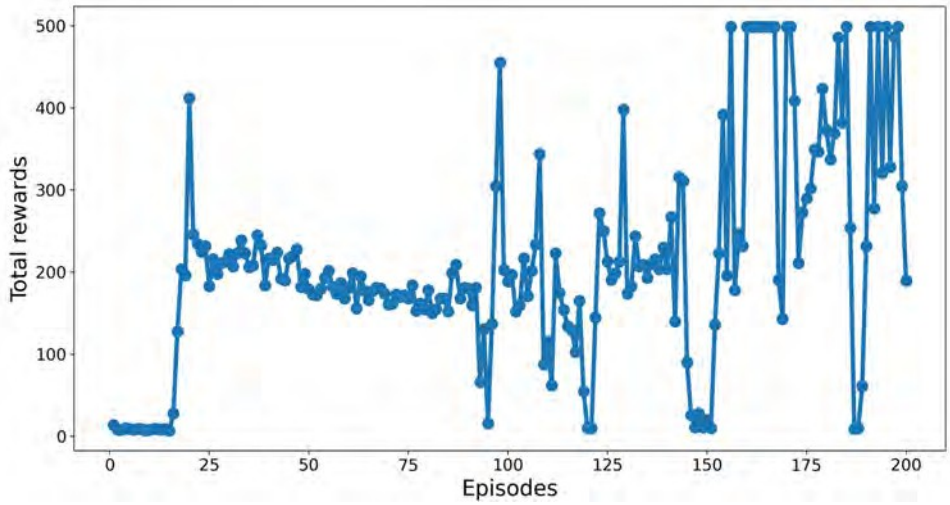
Son olarak, aşağıdaki kodla modeli 200 bölüm için eğitiyoruz ve sonunda plot_learning_history() işlevini kullanarak öğrenme geçmişini görselleştiriyoruz:

```
def plot_learning_history(history):
    fig = plt.figure(1, figsize=(14, 5)) ax
    = f i g . add_subplot(1, 1, 1)
    episodes = np.arange(len(history))+1
    plt.plot(episodes, history, lw=4,
             marker='o', markersize=10)
    ax.tick_params(axis='both', which='major', labelsize=15)
    plt.xlabel('Episodes', size=20)
    plt.ylabel('Toplam ödüller', size=20)
    plt.show()

## Genel ayarlar
EPISODES = 200
batch_size = 32
init_replay_memory_size = 500

if name == '  main ':
    env = gym.make('CartPole-v1')
    agent = DQNAgent(env)
```

```
state = env.reset()
state = np.reshape(state, [1, agent.state_size])
## Yeniden oynatma belleğinin doldurulması
for i in range(init_replay_memory_size):
    action = agent.choose_action(state)
    next_state, reward, done, _ = env.step(action)
    next_state = np.reshape(next_state, [1, agent.state_size])
    agent.remember(Transition(state, action, reward,
                              next_state, done))
    eğer bittiyse:
        state = env.reset()
        state = np.reshape(state, [1, agent.state_size])
    else:
        state = next_state
total_rewards, losses = [], []
for e in range(EPISODES):
    state = env.reset() if
    e %10 == 0:
        env.render()
    state = np.reshape(state, [1, agent.state_size])
    for i  in range(500):
        action = agent.choose_action(state) next_state,
        reward, done, _ = env.step(action) next_state =
        np.reshape(next_state,
                              [1, agent.state_size])
        agent.remember(Transition(state, action, reward,
                                  next_state, done))
        state = next_state if
        e %10 == 0:
            env.render()
        if done:
            total_rewards.append(i)
            print(f'Bölüm: {e}/{EPISODES}, Toplam ödül: {i}') break
        kayıp = agent.replay(batch_size)
        kayıplar.append(kayıp)
plot_learning_history(total_rewards)
```

 alinpikbarbell@gmail.com

Ajanı 200 bölüm boyunca eğittikten sonra, *Şekil 19.13*'te gösterildiği gibi, ajanın gerçekten de zaman içinde toplam ödülleri artırmayı öğrendiğini görüyoruz:



Şekil 19.13: Temsilcinin ödülleri zaman içinde arttı

Bir bölümde elde edilen toplam ödüllerin, temsilcinin direği dengeleyebildiği süreye eşit olduğunu unutmayın. Bu şekilde çizilen öğrenme geçmişi, yaklaşık 30 bölümden sonra ajanın direği nasıl dengeleyeceğini ve 200'den fazla zaman adımı boyunca nasıl tutacağını öğrendiğini göstermektedir.

Bölüm ve kitap özeti

Bu bölümde, en temelden başlayarak RL'deki temel kavramları ve RL'nin karmaşık ortamlarda karar vermeyi nasıl destekleyebileceğini ele aldık.

Etmen-çevre etkileşimleri ve **Markov karar süreçleri (MDP'ler)** hakkında bilgi edindik ve RL problemlerini çözmek için üç ana yaklaşımı ele aldık: dinamik programlama, MC öğrenme ve TD öğrenme. Dinamik programlama algoritmasının çevre dinamikleri hakkında tam bilgiye sahip olunduğunu varsaydığı gerçeğini tartıştık ki bu varsayım çoğu gerçek dünya problemi için tipik olarak doğru değildir.

Ardından, MC ve TD tabanlı algoritmaların bir ajanın çevreyle etkileşime girmesine ve simüle edilmiş bir deneyim oluşturmasına izin vererek nasıl öğrendiğini gördük. Altta yatan teoriyi tartıştıktan sonra, Q-öğrenme algoritmasını, ızgara dünya örneğini çözmek için TD algoritmasının politika dışı bir alt kategorisi olarak uyguladık. Son olarak, büyük veya sürekli durum uzaylarına sahip problemler için kullanılabilecek fonksiyon yaklaşımı ve özellikle derin Q-öğrenme kavramını ele aldık.

Python Makine Öğrenimi'nin bu son bölümünü ve makine öğrenimi ve derin öğrenmeyle ilgili heyecan verici turumuzu beğendiğinizi umuyoruz. Bu kitabın yolculuğu boyunca, bu alanın sunduğu temel konuları ele aldık ve artık gerçek dünyadaki sorunları çözmek için bu teknikleri uygulamaya koymak için iyi bir donanıma sahip olmalısınız.

Bilgisayarlara Verilerden Öğrenme Yeteneği Kazandırma başlıklı *Bölüm 1'deki* yolculuğumuza farklı öğrenme görevleri türlerine kısa bir genel bakışla başladık: denetimli öğrenme, pekiştirmeli öğrenme ve denetimsiz öğrenme.

Daha sonra, *Sınıflandırma için Basit Makine Öğrenimi Algoritmalarının Eğitimi* başlıklı *Bölüm 2'*de basit tek katmanlı NN'lerden başlayarak sınıflandırma *için* kullanabileceğiniz birkaç farklı öğrenme algoritmasını ele aldık.

Bölüm 3, *Scikit-Learn Kullanarak Makine Öğrenimi Sınıflandırıcıları Turu'nda* gelişmiş sınıflandırma *algoritmalarını* tartışmaya devam ettik ve Bölüm *4*, *İyi Eğitim Veri Kümeleri Oluşturma - Veri Önişleme* ve *Bölüm 5, Boyut Azaltma Yoluyla Verileri Sıkıştırma'*da bir makine öğrenimi *işlem* hattının en önemli yönlerini öğrendik.

En gelişmiş algoritmanın bile öğrenmek için aldığı eğitim verilerindeki bilgilerle sınırlı olduğunu unutmayın. Bu nedenle, *Bölüm 6, Model Değerlendirme ve Hiperparametre Ayarlama için En İyi Uygulamaları Öğrenme'd*e, makine öğrenimi uygulamalarında bir başka önemli husus olan tahmine dayalı modeller oluşturmak ve değerlendirmek için en iyi uygulamaları öğrendik.

Tek bir öğrenme algoritması istediğimiz performansı sağlayamıyorsa, bazen bir tahminde bulunmak için bir uzmanlar topluluğu oluşturmak faydalı olabilir. Bu konuyu *Bölüm 7'de, Topluluk Öğrenimi için Farklı Modelleri Birleştirme başlığı* altında inceledik.

Ardından, *Bölüm 8'*de, *Duygu Analizine Makine Öğrenimini Uygulamak*'ta, internetteki sosyal medya platformlarının hakim olduğu modern çağın en popüler ve ilginç veri biçimlerinden birini analiz etmek için makine öğrenimini uyguladık - metin belgeleri.

Çoğunlukla, makine öğreniminin muhtemelen en popüler uygulaması olan sınıflandırma algoritmalarına odaklandık. Ancak, yolculuğumuz burada sona ermedi! *Bölüm 9'*da, *Regresyon Analizi ile Sürekli Hedef Değişkenlerin Tahmini'nde,* sürekli hedef değişkenleri tahmin etmek için regresyon analizine yönelik çeşitli algoritmaları inceledik.

Makine öğreniminin bir diğer heyecan verici alt alanı, eğitim verilerimiz öğrenmek için doğru cevaplarla gelmese bile verilerdeki gizli yapıları bulmamıza yardımcı olabilecek kümeleme analizidir. Bununla *Bölüm 10'*da, *Etiketsiz Verilerle Çalışma - Kümeleme Analizi'nde* çalıştık.

Daha sonra dikkatimizi tüm makine öğrenimi alanındaki en heyecan verici algoritmalardan biri olan yapay sinir ağlarına kaydırdık. *Sıfırdan Çok Katmanlı Yapay Sinir Ağı Uygulaması* başlıklı *11. Bölüm'*de NumPy ile sıfırdan çok katmanlı *bir* algılayıcı uygulayarak başladık.

PyTorch'un derin öğrenme için faydaları, NN modelleri oluşturma sürecini kolaylaştırmak için PyTorch'u kullandığımız, PyTorch Veri Kümesi nesneleriyle çalıştığımız ve bir veri kümesine ön işleme adımlarının nasıl uygulanacağını öğrendiğimiz *Bölüm 12,* PyTorch ile *Sinir Ağı Eğitimini Paralelleştirme*'de açıkça ortaya çıktı.

Bölüm 13, *Going Deeper - PyTorch'un Mekaniği'nde PyTorch'un* mekaniğini daha derinlemesine inceledik ve tensör nesneleri, bir hesaplamanın gradyanlarını hesaplama ve sinir ağı modülü torch.nn dahil olmak üzere PyTorch'un farklı yönlerini ve mekaniğini tartıştık.

Derin Evrişimsel Sinir Ağları ile Görüntülerin Sınıflandırılması başlıklı *14. Bölümde,* görüntü sınıflandırma görevlerindeki yüksek performansları nedeniyle şu anda bilgisayarla görmede yaygın olarak kullanılan evrişimsel sinir ağlarını inceledik.

Bölüm 15, *Tekrarlayan Sinir Ağları Kullanarak Sıralı Verilerin Modellenmesi'*nde, RNN'leri kullanarak *sıralı* modelleme hakkında bilgi edindik.

Bölüm 16, *Dönüştürücüler - Dikkat Mekanizmalarıyla Doğal Dil İşlemeyi İyileştirme'*de, RNN'lerin zayıf yönlerinden birini, yani uzun dizilerle uğraşırken önceki girdi öğelerini hatırlamayı ele almak için dikkat mekanizmasını tanıttık. Daha sonra, kendi kendine dikkat mekanizması etrafında merkezlenen ve büyük ölçekli dil modelleri oluşturmak için son teknolojiyi oluşturan derin öğrenme mimarileri olan çeşitli eski mimarileri inceledik.

 alinpikbarbell@gmail.com

Bölüm 17'de, *Yeni Verilerin Sentezlenmesi için Üretken Çekişmeli Ağlar'da,* GAN'ları kullanarak yeni görüntülerin nasıl üretileceğini gördük ve bu arada oto kodlayıcılar, toplu normalleştirme, aktarılmış konvolüsyon ve Wasserstein GAN'ları hakkında da bilgi edindik.

Önceki bölümler tablo veri kümelerinin yanı sıra metin ve görüntü verilerine odaklanmıştı. *Bölüm 18*'de, Çizge *Yapılı Verilerde Bağımlılıkları Yakalamak için Çizge Sinir Ağları'nda,* sosyal ağlar ve moleküller (kimyasal bileşikler) için yaygın olarak kullanılan veri temsili olan çizge yapılı veriler için derin öğrenmeye odaklandık. Ayrıca, bu tür verilerle uyumlu derin sinir ağları olan sözde çizge sinir ağları hakkında bilgi edindik.

Son olarak, bu bölümde makine öğrenimi görevlerinin ayrı bir kategorisini ele aldık ve bir ödül süreci aracılığıyla çevreleriyle etkileşime girerek öğrenen algoritmaların nasıl geliştirileceğini gördük.

Derin öğrenme üzerine kapsamlı bir çalışma bu kitabın kapsamının çok ötesinde olsa da, derin öğrenme alanındaki en son gelişmeleri takip etmek için ilginizi yeterince çekebildiğimizi umuyoruz.

Makine öğrenimi alanında bir kariyer yapmayı düşünüyorsanız ya da sadece bu alandaki güncel gelişmeleri takip etmek istiyorsanız, bu alanda yayınlanan son literatürü takip etmenizi tavsiye edebiliriz. Aşağıda özellikle faydalı bulduğumuz bazı kaynaklar yer almaktadır:

- Makine öğrenimini öğrenmeye adanmış bir subreddit ve topluluk: https://www.reddit.com/r/learnmachinelearning/
- arXiv ön baskı sunucusuna yüklenen en son makine öğrenimi makalelerinin günlük olarak güncellenen listesi: https://arxiv.org/list/cs.LG/recent
- arXiv üzerine inşa edilmiş bir makale öneri motoru: http://www.arxiv-sanity.com

Son olarak, biz yazarların neler yaptığını bu sitelerden öğrenebilirsiniz:

- Sebastian Raschka: https://sebastianraschka.com
- Hayden Liu: https://www.mlexample.com/
- Vahid Mirjalili: http://vahidmirjalili.com

Bu kitap hakkında herhangi bir sorunuz varsa veya makine öğrenimi hakkında genel ipuçlarına ihtiyacınız varsa bizimle her zaman iletişime geçebilirsiniz.

Kitabımızın Discord alanına katılın

Benzer düşünen insanlarla tanışmak ve 2000'den fazla üyemizle birlikte öğrenmek için Discord topluluğumuza katılın:
https://packt.link/MLwPyTorch

Packt

packt.com

Kişisel gelişiminizi planlamanıza ve kariyerinizi ilerletmenize yardımcı olacak sektörün önde gelen araçlarının yanı sıra 7.000'den fazla kitap ve videoya tam erişim için çevrimiçi dijital kütüphanemize abone olun. Daha fazla bilgi için lütfen web sitemizi ziyaret edin.

Neden abone olalım?

- 4,000'den fazla sektör profesyonelinden pratik e-Kitaplar ve Videolarla öğrenmeye daha az, kodlamaya daha çok zaman ayırın
- Sizin için özel olarak hazırlanmış Beceri Planları ile öğreniminizi geliştirin
- Her ay ücretsiz bir e-kitap veya video alın
- Hayati bilgilere kolay erişim için tamamen aranabilir
- İçeriği kopyalayıp yapıştırın, yazdırın ve yer imlerine ekleyin

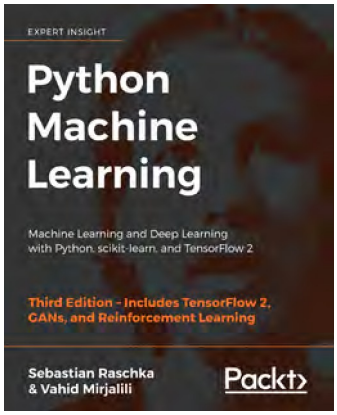
www.packt.com adresinde ayrıca ücretsiz teknik makaleler okuyabilir, çeşitli ücretsiz haber bültenlerine kaydolabilir ve Packt kitapları ve e-Kitaplarında özel indirimler ve teklifler alabilirsiniz.

6587191CA584E
olimpikbarbell@gmail.com

olimpikbarbell@gmail.com

Hoşunuza Gidebilecek Diğer Kitaplar

Bu kitabı beğendiyseniz, Packt'in diğer kitapları da ilginizi çekebilir:



Python Makine Öğrenimi, Üçüncü Baskı

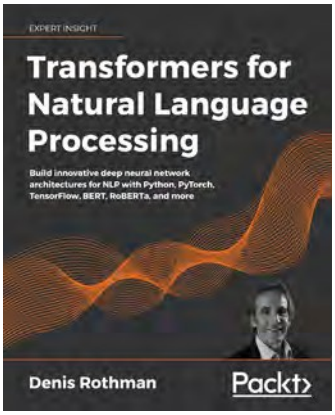
Sebastian Raschka

Vahid Mirjalili

ISBN: 9781789955750

- Makinelerin verilerden 'öğrenmesini' sağlayan çerçeveler, modeller ve tekniklerde uzmanlaşın
- Makine öğrenimi için scikit-learn ve derin öğrenme için TensorFlow kullanın
- Makine öğrenimini görüntü sınıflandırma, duyarlılık analizi, akıllı web uygulamaları ve daha fazlasına uygulayın
- Sinir ağları, GAN'lar ve diğer modelleri oluşturma ve eğitme
- Modelleri değerlendirmek ve ayarlamak için en iyi uygulamaları keşfedin
- Regresyon analizi kullanarak sürekli hedef sonuçları tahmin edin
- Duygu analizini kullanarak metinsel ve sosyal medya verilerini daha derinlemesine inceleyin

6587191CA584E



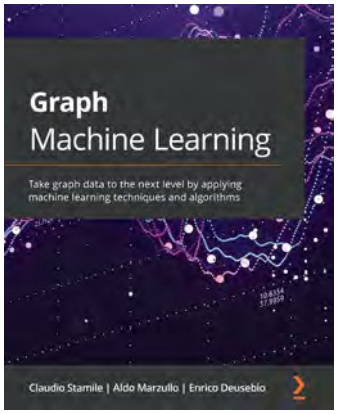
Doğal Dil İşleme için Dönüştürücüler

Denis Rothman

ISBN: 9781800565791

- En yeni ön eğitimli transformatör modellerini kullanın
- Orijinal Transformatör, GPT-2, BERT, T5 ve diğer transformatör modellerinin işleyişini kavrama
- Klasik derin öğrenme modellerinden daha iyi performans gösteren kavramları kullanarak dili anlayan Python programları oluşturun
- Hugging Face, Trax ve AllenNLP dahil olmak üzere çeşitli NLP platformlarını kullanma
- Python, TensorFlow ve Keras programlarını duygu analizi, metin özetleme, konuşma tanıma, makine çevirileri ve daha fazlasına uygulayın
- Üretimdeki kapsamlarını, potansiyellerini ve sınırlarını tanımlamak için anahtar transformatörlerin verimliliğini ölçmek

kitapkurdu.com



Grafik Makine Öğrenimi

Claudio Stamile Aldo

Marzullo Enrico

Deusebio ISBN:

9781800204492

- Grafiklerden özellik çıkarmak için Python komut dosyaları yazma
- Ana grafik temsili öğrenme teknikleri arasında ayrım yapabilme
- Sosyal ağlardan, finansal işlem sistemlerinden ve daha fazlasından veri çıkarma konusunda uzmanlaşın
- Ana denetimsiz ve denetimli grafik gömme tekniklerini uygulama
- Sığ gömme yöntemleri, çizge sinir ağları, çizge düzenleme yöntemleri ve daha fazlası ile tanışın
- Uygulamanızı sorunsuz bir şekilde dağıtın ve ölçeklendirin

6587191CA584E

Packt sizin gibi yazarları arıyor

Packt için bir yazar olmakla ilgileniyorsanız, lütfen authors.packtpub.com adresini ziyaret edin ve hemen başvurun. Tıpkı sizin gibi binlerce geliştirici ve teknoloji profesyoneliyle çalıştık ve onların içgörülerini küresel teknoloji topluluğuyla paylaşmalarına yardımcı olduk. Genel bir başvuru yapabilir, yazar aradığımız belirli bir konu için başvurabilir veya kendi fikrinizi gönderebilirsiniz.

Düşüncelerinizi paylaşın

PyTorch ve Scikit-Learn ile Makine Öğrenimi kitabını bitirdiğinize göre, düşüncelerinizi duymak isteriz! Kitabı Amazon'dan satın aldıysanız, lütfen buraya tıklayarak doğrudan bu kitabın Amazon inceleme sayfasına gidin ve geri bildiriminizi paylaşın veya satın aldığınız sitede bir inceleme bırakın.

Değerlendirmeniz bizim ve teknoloji topluluğu için önemlidir ve mükemmel kalitede içerik sunduğumuzdan emin olmamıza yardımcı olacaktır.

olimpikbarbell@gmail.com

Dizin

Semboller

5×2 çapraz doğrulama

192 7-Zip

URL 248

A

doğruluk

sınıflandırma hatasına karşı 57

eylem-değer fonksiyonu 682 tahmin,

Monte Carlo ile 688 açgözlü

politika, 689'dan hesaplama

aktivasyon fonksiyonu, çok katmanlı sinir ağı için

400'ün seçilmesi

aktivasyon fonksiyonları, torch.nn modülü

referans bağlantısı 406

aktivasyonlar

hesaplama, RNN'lerde 504, 505

AdaBoost

uygulama, scikit-learn ile 233-236

karşılaştırma, gradyan artırma ile 237

AdaBoost tanıma 229

Adam optimizer 479

uyarlanabilir

güçlendirme

zayıf öğrenciler, 229'dan yararlanma

çalışma 229-233

Uyarlanabilir Doğrusal Nöron (Adaline) 35-37, 278

algoritma 337

uygulama, lojistik regresyon için algoritmaya dönüştürme 66-68

Python'da uygulama 39-43

İleri grafik sinir ağlari literatürü

işaretçiler 669, 670

ajan 6, 674, 675 aglomeratif

kümeleme

scikit-learn aracılığıyla uygulama 327, 328

Yapay zeka kışları

referans bağlantısı 336

Ames Konut veri seti 272

veri çerçevesine yükleme 272-274

doğrusal olmayan ilişkiler, modelleme 297-299

Anaconda 15

referans bağlantısı 15

Anaconda Python dağıtımı

15, 16 kullanarak

yapay zeka (AI) 1, 19, 336, 452 yapay sinir

ağları

EĞİTİM 360

karmaşık fonksiyonları modellemek için kullanılır 335, 336

yapay nöron 20

dikkat mekanizması

RNN'ler için 540, 542

dikkat ağirliklar

bilgi işlem 544

otomatik kodlayıcılar 590, 591
otomatik farklılaştırma 363, 416
otoregresyon 534
ortalama havuzlama 463

B

geri yayılım 350, 362, 363
sinir ağları, 363-367 kaynaklar
aracılığıyla eğitim 343
torbalama 95, 223, 224
uygulamak, Wine veri setindeki örnekleri sınıflandırmak 225-228
model toplulukları için
225 **kelime torbası modeli**
250 temel grafik
konvolüsyonu
644-647'nin uygulanması
Temel Doğrusal Cebir Alt Programları (BLAS)
29 toplu gradyan inişi 39
BatchNorm 614-616
Bellman denklemi 676
dinamik programlama 684
önyargı ünitesi 21
önyargı-varyans değiş tokuşu 73
Çift Yönlü ve Otomatik Regresif Transformatör (BART) 561, 572-574
Transformatörlerden Çift Yönlü Kodlayıcı Temsilleri (BERT) 561, 569
çift yönlü ön eğitim 569-571 model,
PyTorch'ta ince ayar 574
çift yönlü RNN 524, 525
girdiler, işleme 543
daha büyük veri
birlikte çalışma 261, 262
ikili çapraz entropi 471
binom katsayısı 207
95'e yükseltme
sınır noktası 328
Meme Kanseri Wisconsin veri kümesi
yükleme 172, 173
172 elde etmek
Broyden-Fletcher-Goldfarb-Shanno (BFGS) 71

C

aday değer 512
otomobilin yakıt verimliliği tahmin projesi
DNN regresyon modeli, eğitim 435, 436
özellik sütunları, 431-434 ile çalışma
Katalizör
URL 439
CatBoost
URL 244
kategorik çapraz entropi 471 kategorik
veri
pandas 111 ile kodlama
kullanım 111
CelebA veri kümesi
alternatif yollar, 483 yüklemesini
indirmek için 483
hücre durumu 511
merkezi işlem birimleri (CPU'lar) 370
centroid 306
kanallar 464
karakter düzeyinde dil modellemesi, PyTorch 525, 526
veri kümesi, işleme 526-531
metin pasajları, 533-537 oluşturma
karakter düzeyinde RNN modeli
bina 531, 532
sınıflandırma
sınıf etiketleri, tahmin 4, 5
sınıflandırma algoritması
53'ün seçilmesi
sınıflandırma hatası 88
doğruluğa karşı 57

sınıflandırma modeli
hassasiyet, optimize etme 195-197
hatırlama, optimize etme 195-197
sınıflandırma görevi 4
sınıflandırıcılar
oy çokluğuyla birleştirilmesi 209
sınıf dengesizliği
201-203'ü yönetmek
sınıf etiketleri
kodlama 112, 113
sınıf üyelik olasılıkları
karar ağaçlarından 214
sınıf olasılıkları
softmax fonksiyonu aracılığıyla çok sınıflı sınıflandırmada tahmin 402
lojistik regresyon yoluyla modelleme 59
küme uyumu 314
küme ataleti 308
kümeleme 305
alt grupları bulmak için kullanılır 8
kümeler
gruplama 320, 321
hiyerarşik ağaç olarak düzenleme 319
küme ayırma 314 CNN
katmanları
yapılandırma, PyTorch'ta 476
CNN gülümseme sınıflandırıcısı
EĞİTİM 490-497
eş doğrusallık 159
renk kanalları 464
yaygın mimariler uygulamalar özel
katmanlar, PyTorch'ta yazma 426-430
kayıp fonksiyonu, seçme 418, 419
model oluşturma, nn.Module ile esnek hale getirme 424-426
nn.Sıralı tabanlı model, 417, 418'in uygulanması
torch.nn modülü 417 aracılığıyla basitleştirme
XOR sınıflandırma problemi, 419-423 çözme
tam bağlantı 320 karmaşık
fonksiyon
modelleme, yapay sinir sistemi ile ağlar 335, 336
karmaşık NN'ler
uygulamalar 336
hesaplama grafikleri, PyTorch 410
bilgisayar işlemcileri
performans mücadelesi 369, 370
conda-forge
URL 16
koşullu GAN (cGAN) 635
koşullu çıkarım 593
koşullu olasılıklar 60-63
karışıklık matrisi
okuma 193-195
bağlam katıştırmaları
kodlama, çok kafalı dikkat yoluyla 554-558
bağlam vektörleri
çıktılar, 543, 544'ten üretme
devam eden görev 680
öğrenme yakınsaması 36
konveks optimizasyon
algoritmalar 71
konvolüsyonel GAN'lar
mimari tasarım hususları 618 kullanılan, sentezlenmiş görüntü kalitesi 612
konvolüsyonel sinir ağları (CNN'ler) 451, 452
oluşturma, PyTorch'ta 477-481
grafikler, referans bağlantısı 666
uygulama 464
torch.nn modülü ile uygulama 476
gülümseme sınıflandırması, yüz görüntülerinden 482
konvolüsyon çıktısı
boyut, belirleme 458, 459

çekirdek nokta
328 korelasyon
matrisi
276-278 kullanarak
çapraz korelasyon
456 cuML
referans bağlantı 102
boyutluluk laneti 102, 132, 517 özel
katmanlar
yazma, PyTorch içinde 424-430

D

Veri
ön işleme 13
sentezleme, üretken modellerle 592, 593
veri artırımı 482-484, 488 veri
çerçevesi
Ames Konut veri kümesi, 272-274'e yükleme
DataLoader
655-658 hazırlanıyor
veri bilimi
16, 17 numaralı paketler
veri kümesi
ayrı eğitim ve test veri kümelerine bölümleme 117, 118
veri görselleştirme
t-dağıtımlı stokastik komşu gömme yoluyla 165-168
DBSCAN
yüksek yoğunluklu bölgeler, yer belirleme 328-333
DCGAN modeli
WGAN-GP'yi uygulamak için eğitim 629-632
karar bölgeleri 57
karar ağacı 86, 87, 119
Bina 92-95
sınıf üyelik olasılıkları 214 **karar**
ağacı regresyonu 300 kod çözücü
558
dekoder ağı 590
dekonvolüsyon 613
dekonvolüsyon işlevi 613 derin
CNN
PyTorch ile uygulama 473
derin evrişimli GAN (DCGAN) 612
DeeperGCN
referans bağlantısı 669
derin öğrenme 339
derin NN 339
derin Q-öğrenme 706
algoritması, uygulama 710, 714 **derin**
Q-ağ (DQN) 706 dendrogramlar 319
ısı haritasına ekleme 326, 327
yoğunluk tabanlı kümeleme 305
DiffPool 668
boyutsallık azaltma 128, 591
veri sıkıştırma için 8
yönlendirilmiş asiklik grafik
(DAG) 410 yönlendirilmiş
grafikler 639
indirim faktörü 680 ayrık
konvolüsyonlar
2D'de 459-461
tek boyutta 454-456
gerçekleştirme 454
diskriminatör 594
diskriminatör ağları
616-624'ün uygulanması
mesafe matrisi
hiyerarşik kümeleme, 321-324 üzerinde performans
DistilBERT model 575 dağıtım
benzerlik ölçüleri 624-626
DNN regresyon modeli
EĞİTİM 435

belge sınıflandırması
lojistik regresyon modeli, 258-260 için eğitim
belgeler
işleme, belirteçlere dönüştürme 256, 257
nokta çarpımı 21
DQN modeli
Q-öğrenme algoritmasına 706 göre eğitim
bırakma
NN, düzenli hale getirme 469, 470
dinamik programlama 677, 685
değer fonksiyonu, Bellman
denklemi ile 686 tahmin etme
684

E

eigendecomposition 141, 144
özdeğerler 141
referans bağlantı 141
özvektörler 141
referans bağlantı 141
elastik ağ 292
dirsek yöntemi 306
optimum küme sayısını bulmak için
kullanma 313, 314
eleman-bilge çarpım 511
eleman bazında toplama 511
ELMo (Dil Modellerinden Gömüler) 562 gömme
avantajları, tek-sıcak kodlamaya göre 518
EM mesafesi
avantajları 626
GAN'lar için kullanma
627
kodlayıcı bloğu 554
kodlayıcı ağı 590
kodlama şemaları
nominal veriler için 116
topluluk 95
bina, istifleme ile 223
kayıp gradyanlarına dayalı eğitim 237
topluluk sınıflandırıcı
217-223'ü değerlendirmek
ayarlama 217-223
topluluk yöntemleri 205-208
entropi 88
çevre 675
bölüm 674, 680
epizodik görev 679
epsilon 167
hata fonksiyonu 12
tahmini değer
fonksiyonu
politika, iyileştirme 686
tahmin ediciler 110
beklenti-maksimizasyonu (EM)
referans bağlantı 266
açıklanan varyans 146
patlayan gradyan 510
sömürü 676
keşif 676
keşifsel veri analizi (EDA) 274

F

F1 puanı 193
Facebooks AI Research lab (FAIR) 371
yanlış negatif (FN) 194
yanlış pozitif (FP) 194
yanlış pozitif oran (FPR)
195 fastai
URL 439
özellik 12 özellik
sütunları
431-434 ile çalışmak
özellik katkıları
152-154'ü değerlendirmek
özellik çıkarma 128, 139

özellik hiyerarşisi 452 özellik

önemi

değerlendirme, rastgele ormanlar ile 134-136

özellik haritaları 453

özellik normalizasyonu 74

Özellikler 10

eksik değerlerle eleme 107, 108

122'nin seçilmesi

özellik ölçeklendirme 119, 120

gradyan inişi, iyileştirme 43-45

özellik seçimi 128 özellik

seçimi algoritmaları

scikit-learn'de 134

özellik uzayı

örnekler 161

özellik alt uzayı

doğrusal diskriminantlar, 158-160 için seçim

özellik dönüşümü 146-148 özellik

vektörleri

kelimeler, 250, 251'e dönüştürme

ileri besleme 342

birkaç atışla öğrenme 565, 566

filtre 454

Fisher'ın LDA'sı 155

zemin operasyonu 458

unut kapısı 511 ileri

yayılım

sinir ağı, 340, 341 aracılığıyla etkinleştirme

kesirli kademeli konvolüsyon 613 tam

yığın gradyan inişi 39

tam bağlı katmanlar 452

bulanıklık katsayısı 312

bulanık kümeleme 311

bulanık C-ortalamalar (FCM)

algoritması 311 bulanık k-ortalamalar

311

G

GAN uygulamaları 635

geçitli tekrarlayan birim (GRU) 512

kapı 511

Gauss çekirdeği 83

genelleme hatası 14

genelleme performansı

tahmini 180

genelleştirilmiş politika yinelemesi (GPI) 687

üretken düşman ağları (GAN'lar) 589, 590

ayırıcı ağlar, 600-603 ayırıcı ağların uygulanması, kayıp fonksiyonu 594, 595 EM mesafesi, 627 kullanımı

jeneratör ağları, uygulama 600-603 jeneratör ağları, kayıp fonksiyonu 594, 595

596'yı uygulamak

model, eğitim 605-612

model eğitimi, Google Colab 596-599 eğitim

veri kümesi üzerinde, 604, 605'in tanımlanması

kullanılan, örnek oluşturmak için 593

üretken model 593

verilerin sentezlenmesi için 592, 593

Üretken Ön Eğitimli Dönüştürücü (GPT) 561

etiketlenmemiş veri, 563-565'ten yararlanma

jeneratör 593 jeneratör

ağları

616-624'ün uygulanması

Gini safsızlığı 88

küresel yorumlayıcı kilidi (GIL) 370

küresel havuzlama katmanı

değişkenliklerle başa çıkmak için grafik boyutları 652-654

Google Colab

referans bağlantısı 597

GPT-1 564, 565

GPT-2 565
referanslar 568
kullanarak, metin oluşturmak için 566-568
GPT-3 565, 566
gradyan artırma 237 AdaBoost,
sınıflandırma için 237 ile
karşılaştırma 241-243
regresyon için 238
gradyan artırma algoritması 237
sınıflandırma için 239, 240
degrade kırpma 510
gradyan inişi (GD) 56, 278
iyileştirme, özellik ölçeklendirme yoluyla
43-45 kayıp fonksiyonları, en aza indirme
37-39 regresyon, regresyon için çözme
parametreler 279-283
gradyan inişi optimizasyon algoritması 119
gradyan cezası (GP) 628
gradyanlar
çekişmeli örnekler 416
hesaplama, otomatik farklılaştırma yoluyla 415
kayıp gradyanları, hesaplama 415, 416
Grafik
PyTorch 411, 412'de oluşturma
grafik konvolüsyonları 641, 642
temel grafik konvolüsyonu, 644-
647'nin uygulanması
nedenler, kullanım için 642, 643
grafik işlem birimleri (GPU'lar) 410 grafik
işlem birimleri (GPU'lar) 368-370, 596
grafik Laplacian 665
PyTorch'ta **çizge sinir ağı (GNN)**
uygulaması 648 PyTorch ile uygulama
Geometrik
kütüphane 659-665
katmanlar 665
son gelişmeler 665
GraphNorm 669
referans bağlantısı 668
grafik normalleştirme tekniği 668, 669
Grafikler 638
yönlendirilmiş grafikler 639
etiketli grafikler 640
moleküller, 640 yönlendirilmemiş
çizge olarak temsil 638, 639
açgözlü algoritmalar
129 açgözlü politika
hesaplama, eylem-değer fonksiyonundan 689
ızgara araması
hiperparametreler, ayarlama 186
makine öğrenimi modelleri, ince ayar 185
şebeke dünyası ortamı
OpenAI Spor Salonu 694-701'de uygulama
şebeke dünyası sorunu
çözme, Q-öğrenme ile 701-706

H

el yazısı rakamlar, 343 MNIST veri
kümesinin **sınıflandırılması,** 343-
346 MNIST veri kümesinin elde
edilmesi, 343-346 çok katmanlı
algılayıcı (MLP) hazırlanması,
347-351'in uygulanması
sinir ağı performansı,
değerlendirme 357-360
sinir ağı eğitim döngüsü, kodlama
352-356
sert kümeleme 311
ısı haritası 325
dendrogramlar, 326, 327'ye ekleme
gizli nüks
çıktı tekrarına karşı 506-508
gizli yapılar
keşfetme, denetimsiz öğrenme ile 7
hiyerarşik kümeleme 306, 319
aglomeratif 320

bölücü 320
gerçekleştirme, mesafe matrisi üzerinde 321-324

Farklılaştırılabilir Havuzlama ile Hiyerarşik Grafik Gösterimi Öğrenimi
referans bağlantısı 668

hiyerarşik ağaç
kümeleri, 319 **yüksek**

varyanslı 122

HistGradientBoostingClassifier

olarak düzenler.
URL 244

tutma yöntemi 175, 176 insan

görsel korteksi 452 hiperbolik

tanjant (tanh) 403
çıkış spektrumunu genişletmek için kullanılır 403, 404

hiperopt
referans bağlantısı 191

hyperopt-sklearn
referans bağlantısı 191

hiperparametre konfigürasyonları
keşfetme, rastgele arama ile 187, 188

hiperparametreler 42, 175
ayarlama, ızgara arama yoluyla 186

I

IID (bağımsız ve özdeş dağılımlı) 707

görüntü dönüşümleri 486-489

IMDb film inceleme verileri
metin işleme için hazırlama 247, 248

IMDb film inceleme veri kümesi
yükleme 575, 576
simgeleştirme 577, 578

anında ödül 680

safsızlık 88

bağımsız ve özdeş dağılımlı (IID) 500

bilgi kazancı (IG) 87, 300
maksimize etme 88-92

ilk küme merkezleri
yerleştirme, k-means++ ile 310, 311

inliers 285

giriş kapısı 512

giriş boru hattı
bina, PyTorch'ta 378

girdiler
dolgu, çıktı özellik haritalarının boyutunu kontrol etmek için 457
çift yönlü RNN 543 ile işleme

akıllı makineler
veriyi bilgiye dönüştürmek için inşa etmek 1, 2

iç kovaryans kayması 616 Iris

veri seti 54
yükleme 31
perceptron modeli, eğitim 29-35
referans bağlantı 9

J

Jensen-Shannon (JS) 625

Jupyter Notebook 597

Jupyter Notebook GUI

referans bağlantısı 597

K

Kantorovich-Rubinstein ikiliği 627

Çekirdek 454

çekirdek fonksiyonu 83

çekirdek yöntemleri 81

çekirdek SVM 80
doğrusal olmayan problemleri çözmek için kullanılır80

çekirdek numarası 83
kullanarak, yüksek boyutlu uzayda ayırıcı hiper düzlemleri bulmak için 82-86

 olimpikbarbell@gmail.com

k-kat çapraz doğrulama 176-179

K-Lipschitz sürekli 627 k-

ortalamalar

nesneler, benzerliğe göre gruplama 305

k-means++

başlangıç küme merkezleri, yerleştirme 310, 311

k-ortalamalar kümeleme

scikit-learn ile 305-310

k-en yakın komşular (KNN) 98-102, 119

Kullback-Leibler (KL) 625

L

L1 düzenlemesi 123

seyrek çözümler 125-128

seyreklik 125

L2 düzenlemesi 74, 122

geometrik yorumlama 123, 124

NN, ağırlık azalmasına

karşı 467 düzenli hale

getirme 469

etiketli grafik 640

dil modeli 558

dil modelleme 525

dil çevirisi 561 büyük

ölçekli dil modelleri

etiketlenmemiş verilerden yararlanarak oluşturma 561

büyük ölçekli makine öğrenimi

45 Latent Dirichlet Allocation

metin belgeleri, ayrıştırma 264

konu modelleme 264

scikit-learn ile 265-268

gizli vektör 590

katman normalleştirme

mekanizması 560 tembel öğrenici

98

Sızdıran ReLU 601

öğrenme eğrileri 180

yanlılık ve varyans sorunları, teşhis 180-183

öğrenme oranı 23, 337
en küçük mutlak küçültme ve seçme operatörü (LASSO) 292, 293
dışarıda bırakarak çapraz doğrulama (LOOCV) yöntemi 178
LightGBM
URL 244
Doğrusal Cebir Paketi (LAPACK) 29
doğrusal diskriminant analizi (LDA) 591
denetimli veri sıkıştırma 154
temel bileşen analizine karşı 154, 155
scikit-learn aracılığıyla 162, 163
ÇALIŞMALAR 156
doğrusal ayırıcılar
seçme, özellik alt uzayı için 158-160
doğrusal en küçük kareler 278
doğrusal olarak ayrıştırılamayan veri çekirdeği yöntemleri 80, 81
doğrusal regresyon 5, 269
doğrusal regresyon modeli
performans, değerlendirme 288-292
dönüş, eğriye 294
bağlantı matrisi 322
bağlantılar 320
Lipschitz sürekliliği 627
yükler 152
yerel alıcı alan 453
lojistik fonksiyon 400, 402
lojistik kayıp fonksiyonu
model ağırlıkları, 63-65 aracılığıyla öğrenme
lojistik regresyon 60-63
Adaline uygulaması, algoritmaya dönüştürme 66-68
sınıf olasılıkları, birden fazla sınıf için 59 aracılığıyla modelleme 60
Kaynak 76
destek vektör makinesine (SVM'ler) karşı 79

lojistik regresyon modeli
eğitim, belge sınıflandırması için 258-260
eğitim, scikit-learn ile 70-72
lojistik sigmoid fonksiyonu
61 log-likelihood
fonksiyonu 63 uzun
menzilli etkileşimler
zorluklar 509, 510
uzun kısa süreli bellek hücreleri 511, 512
kayıp fonksiyonu 12
bilgi işlem 360-362
sınıflandırma için 471-473
minimize etme, gradyan inişi ile 37-39
düşük seviyeli özellikler 452

M

makine öğrenimi 11
GPU destekli uygulamalar 102 modeller,
ızgara arama yoluyla ince ayar 185 paketler
16, 17
Python, 14 takviyeli
öğrenme için kullanma 2
denetimli öğrenme 2
sistemler, iş akışı 12
denetimsiz öğrenme 2
çoğunluk oyu 95
sınıflandırıcılar, 209 aracılığıyla birleştirme
çoğunluk oy sınıflandırıcı
209-213'ün uygulanması
çoğunluk oylama ilkesi 205
tahminler yapmak için kullanmak 214-217
Markov karar süreçleri (MDP'ler) 676, 677, 685
matematiksel formülasyon 677, 678
Markov süreci
görselleştirme 679
maskeli dikkat 558
maskelenmiş dil modeli (MLM) 570 matris
çarpımı 41

6587191CA584E

matris transpozesi 21
Matthews korelasyon katsayısı (MCC) 193
maksimum marj sınıflandırması
destek vektör makineleri (SVM) ile 76
maksimum marj
sezgisi 77
maksimum
havuzlama 463
MC kontrolü
kullanılan, optimal
politikayı bulmak için 688
McCulloch-Pitts (MCP)
nöronu 19 McCulloch-
Pitts nöron modeli 336
ortalama mutlak hata
(MAE) 291 ortalama
imputasyon 108
karesel hataların
ortalaması (MSE) 337
ortalama havuzlama
463
ortalama karesel hata türevi 38
ortalama karesel hata (MSE) 37, 279,
290-392 medyan mutlak sapma 286
medoid 306
bellek temelli yaklaşımlar
avantajlar 100
dezavantajlar 100
bellek hücresi 511
mesaj iletme çerçevesi 647
mini parti ayrımcılığı
634 mini parti
gradyan inişi 47
Miniconda 15
referans bağlantısı 15
Miniforge 15
referans bağlantısı 15
min-max normalleştirme 121
min-maks ölçekleme 120
yansıtılmış projeksiyonlar 147
eksik veri
105'i yönetmek
eksik değerler
özellikler, ortadan kaldırma 107, 108

tanımlama, tablo verilerinde 106
imputing 108, 109
eğitim örnekleri, ortadan kaldırma 107, 108

MNIST veri kümesi
yükleme 474, 475

MNIST el yazısı rakamları sınıflandırma projesi 436-439

mod çökmesi 633, 634

model tabanlı RL 678

model toplulukları
torbalama ile 225

modelsiz RL 678

model

parametreleri
depolama, kullanılan PyTorch tensör nesneleri 412-414
güncelleme, kullanılan PyTorch tensör nesneleri 412-414

modeller
14'ü değerlendirmek

model seçimi 175

model ağırlıkları
öğrenme, lojistik kayıp fonksiyonu aracılığıyla 63-65

moleküller
grafikler olarak 640

Monte Carlo (MC) 678, 684
eylem-değer fonksiyonu tahmini 688
durum-değer fonksiyonu tahmini 688

film veri kümesi
ön işleme 248, 249

film inceleme veri kümesi
elde edilmesi 248

MsgNorm 669

çok sınıflı sınıflandırma 5

çok kafalı dikkat
bağlam katıştırmaları, kodlama 554-558

çok başlı kendi kendine dikkat 554

çok katmanlı CNN mimarisi 473, 474

çok katmanlı sinir ağı
aktivasyon fonksiyonu, 400 seçme

çok katmanlı sinir ağı mimarisi 338-340

çok katmanlı algılayıcı (MLP) 338, 370, 452, 501
gizli katmanlar, 339 katman
ekleme 339

multinomial lojistik regresyon 60 çoklu

karar ağaçları
rastgele ormanlar aracılığıyla birleştirme 95-98

çoklu giriş kanalları 465 çoklu

doğrusal regresyon 271 çoklu

işlem
n_jobs parametresi aracılığıyla 259

MurmurHash
URL 262

N

naïve Bayes sınıflandırıcı 260 Doğal

Dil Araç Seti (NLTK)
URL 256

iç içe çapraz doğrulama 191
algoritma seçimi 192, 193

NetworkX
yükleme 644
URL 644

Kuantum Kimyası için Sinirsel Mesaj İletimi
referans bağlantısı 647

sinir ağları (NN'ler) 589
etkinleştirme, ileri yayılım yoluyla 340-342 yakınsama
367, 368
uygulama 368
eğitim, geriye yayılım yoluyla 363, 364

sonraki kelime tahmini 561

n-gram modelleri 252

nn.Embedding
referans bağlantısı 434

NN modeli
PyTorch 389'da bina

nn.Module 424
model oluşturma, 424-426 ile esnek hale getirme
nn.Sıralı
referans linki 417
NodeNetwork
grafik konvolüsyon katmanı, kodlama 650, 651
tahminlerde bulunmak için kullanmak 658, 659
NodeNetwork modeli
649'un tanımlanması
düğümler 638
bedava öğle yemeği teoremi
yok 53 gürültü noktası 328
nominal özellikler 111
one-hot kodlama, 113-115 üzerinde performans
YokTip hatası 263
doğrusal olmayan boyut indirgeme 164, 165
doğrusal olmayan problemler
çözme, çekirdek SVM ile 80
doğrusal olmayan ilişkiler
yönetme, rastgele ormanlar ile 299
modelleme, Ames Konut veri setinde 297-299
örtüşmeyen havuzlama
örtüşen havuzlamaya karşı 464
parametrik olmayan modeller
parametrik modellere karşı 99
normalleştirme 120
NumPy dizi indeksleme
28

O

amaç fonksiyonu 37
nesne yönelimli algılayıcı API 25-29
nesneler
gruplama, benzerliğe göre 305
politika dışı Q-öğrenme 685
politika dışı TD kontrolü (Q-öğrenme) 691
ofsetler 270

tek seferlik kodlama
performans, nominal özellikler üzerine 113-115
bir-sıcak vektör temsili
340 bir-karşı-hepsi (OvA)
tekniği 340
çok sınıflı sınıflandırma için 30
bire karşı dinlenme (OvR) 56, 126
bire karşı herkes (OvA)
sınıflandırması 200
çevrimiçi öğrenme 46
on-policy SARSA (state-action-reward-state- action) 685
politika üzerinde TD
kontrolü (SARSA) 691
OpenAI Spor Salonu
692
ortamlar 692, 693
ızgara dünya ortamı, uygulama 694-701
şebeke dünyası
örneği 694 araç
seti 692
URL 692
fikir madenciliği
247 optimal
politika
MC kontrolü ile bulgu 688
sıralı özellikler 111
kodlama 116
haritalama 111, 112
sıradan en küçük kareler doğrusal regresyon modeli
278'in uygulanması
sıradan en küçük
kareler (OLS) 278
çekirdek dışı öğrenme
261
çıkış
kapısı
512 çıkış tekrarı
gizli nükse karşı 506-508
çıkışlar
üretme, bağlam vektörlerinden 543, 544
çıktı spektrumu
genişletme, hiperbolik tanjant ile (tanh) 403-405
aşırı tamamlanmış otomatik kodlayıcı 591

aşırı uyum 57, 122
düzenlileştirme yoluyla mücadele 73-76
örtüşen havuzlama
örtüşmeyen havuzlamaya karşı 464

P

paket yöneticisi
15, 16 kullanarak
dolgu 455
modlar 457
pandalar
kategorik veriler, kodlama 111
pandas DataFrame
veri işleme 107
parametrik modeller
parametrik olmayan modellere karşı 99
ayrıştırıcı modülü
referans bağlantısı 255
Pearson Momentler Çarpımı Korelasyon Katsayısı 276
perceptron
ayarlar 59
perceptron yakınsaması 35
perceptron öğrenme algoritması
Python 25'te uygulama
perceptron öğrenme kuralı 22-24
perceptron modeli
eğitim, Iris veri kümesi üzerinde 29-35
performans değerlendírme metrîklerí
193 permütasyon değíşmezlíğí 642
boru hatti
transformatörler ve tahmin ediciler, 173, 174'ün birleştirilmesi
iş akışları, kolaylaştırma 171
çoklu oylama 205, 209
POLİTİKA 682
tahmini değer fonksiyonu ile iyileştirme 686
politika değerlendirmesi 686
politika geliştirme 689
politika yinelemesi 687
polinom regresyon 294
polinom terimler
scikit-learn ile ekleme 294-296
havuzlama
avantaj 463
havuzlama katmanları 453, 667, 668
havuzlama boyutu 463
Porter stemmer algoritması 256
Porter stemming algoritması 257
konumsal kodlamalar 559, 560
kesinlik-geri çağırma eğrileri 198
tahmin edilen sınıf etiketi 23
tahminler
denetimli gelecek hakkında öğrenme 3, 4
oy çokluğu ilkesi ile karar alma 214-217
tahmine dayalı model
13, 14'ün seçilmesi
Eğitim 13
önceden eğitilmiş BERT modeli
ince ayar 579-582
yükleme 578, 579
temel bileşen analizi (PCA) 173, 591
scikit-learn'de 149-152
denetimsiz boyutluluk azaltma 139-141
doğrusal diskriminant analizine karşı 154, 155
temel bileşenler
çıkarma 142, 143
projeler
otomobilin yakıt verimliliği tahmin projesi 431
MNIST el yazısı rakamları sınıflandırma 436-439
prototip tabanlı kümeleme 305

budama

k 87

Python
Adaline, 39-43'te uygulanıyor
yükleme, Python Paket Dizini'nden 14
URL 14
kullanarak, makine öğrenimi için 14

Python İlerleme Göstergesi (PyPrind)
referans bağlantı 248

PyTorch 369, 371
BERT modeli, ayarlama 574
karakter düzeyinde dil modelleme 525
CNN, 477-481 oluşturma
CNN katmanları, yapılandırma
476 hesaplama grafikleri 410
özel katmanlar, yazma 426-430
özellikler 410
grafik, 411, 412 grafik sinir ağı
(GNN) oluşturma,
648'in uygulanması
yükleme 372, 373
öğrenme 372
referans bağlantısı 377
RNN'ler, dizi modelleme için uygulama
513
tensör nesneleri 412
tensörler, oluşturma 373
eğitim performansı 369
URL 368
derin CNN'i uygulamak için kullanılır 473

İlaç keşfi için **PyTorch**
Geometrik kütüphane 665
kurulum bağlantısı 659
yükleme 659
kullanılan, GNN'yi uygulamak için 659-665

PyTorch-Ignite
URL 439

PyTorch, girdi boru hatları
toplu veri yükleyici, 380-382 oluşturma
378
veri kümesi, yerel depolama diskindeki
dosyadan oluşturma 382-385
veri kümesi, torchvision.datasets
kütüphanesinden getirme 386-389
PyTorch DataLoader, tensörden oluşturma 378
tekrar veri yükleyicisi, oluşturma 380-382
shuffle veri yükleyicisi, 380-382 oluşturma
tensör, ortak veri kümesinde birleştirme 379, 380

PyTorch Yıldırım
veri yükleyicileri, kurulum 443, 444
440'ın kurulumu
model, kurulum 440, 442
URL 439

PyTorch Yıldırım Eğitmeni sınıfı
model, eğitim 444, 445

PyTorch sinir ağı modülü (torch.nn) 390 PyTorch,

NN modeli
doğrusal regresyon modeli, 390-394 oluşturma
torch.nn modülü aracılığıyla model eğitimi 394,
395 torch.optim aracılığıyla model eğitimi
modül 394, 395
çok katmanlı algılayıcı, Iris çiçek veri kümesi
için oluşturma 395-398
eğitilmiş model, test veri kümesi üzerinde
değerlendirme 398 eğitilmiş model, yeniden
yükleme 399
eğitimli model, 399 tasarruf

Q

Q-öğrenme
ızgara dünya problemi, çözüm 701-706

QM9 veri seti 661
referans bağlantısı 661

R

radyal tabanlı fonksiyon (RBF) çekirdeği

83 radyal tabanlı fonksiyon (RBF)

çekirdeği SVM 186 rastgele orman

regresyonu 301-304 rastgele ormanlar

95, 119

özellik önemi, değerlendirme 134-136

çoklu karar ağaçları, 95-98 aracılığıyla
birleştirme doğrusal olmayan ilişkiler, yönetme 299

rastgele arama
hiperparametre konfigürasyonları, 187, 188'i keşfetmek

RANdom SAmple Consensus (RANSAC)
sağlam regresyon modeli, uydurma 285-288

rastgele örnekleme
yedek olmadan 96
yedek 96 ile

ham terim frekansları 251

alıcı işletim karakteristiği eğri altındaki alan (ROC AUC) 212

alıcı işletim karakteristiği (ROC)
çizim 198, 200

düzeltilmiş doğrusal birim (ReLU) 405

tekrarlayan kenar 503

tekrarlayan sinir ağları (RNN'ler)
aktivasyonlar, hesaplama 504, 505
dikkat mekanizması 540, 542
veri akışı 502-504
dizi modelleme için 502

regresyon 4, 5
sürekli sonuçlar, tahmin etme 5, 6

regresyon analizi 5

regresyon çizgisi 270

düzenli ifadeler 255

düzenlileştirme 74

düzenlileştirme parametresi 74

düzenlileştirilmiş yöntemler
kullanarak, regresyon için 292

pekiştirmeli öğrenme 2, 6, 674, 675
algoritmalar 684
Örnek 6
Monte Carlo 687 ile etkileşimli problemleri 6, 7 çözmek için kullanılır

yeniden modül
referans bağlantısı 255

yeniden oynatma belleği 707, 708

artıklar 270

kaynak verimli hiperparametre arama
ardışık yarılanma ile 189, 190

dönüş 680-683

ödül 683

ödül sinyali 6, 675

ridge regresyon 293

RL algoritması
691'in uygulanması

RNN modeli
Duygu analizi görevi için bina 521-523

RNN'ler, dizi modelleme için
PyTorch 513 içinde

Eğri altındaki ROC alanı (ROC AUC) 198

S

doygunluk 595

ölçeklendirilmiş nokta çarpımı

dikkati 549 dağılım matrisleri
bilgi işlem 156-158

scatterplot matrisi 274, 275

bilimsel hesaplama
16, 17 numaralı paketler

bilimsel hesaplama yığını, Python
Referanslar 26

scikit-fuzzy paketi
referans bağlantı 313

scikit-learn
aglomeratif kümeleme, 327, 328 aracılığıyla uygulama
alternatif uygulamalar 79
katsayısı, regresyon modelinin tahmini 283 k-ortalamalar kümeleme 305-310
LDA 162, 163
perceptron, eğitim 54-59 polinom
terimleri, ekleme 294-296
temel bileşen analizi 149-151

AdaBoost uygulamak için kullanılır
233-236 lojistik regresyon eğitimi için
kullanılır
model 70-72
scikit-learn tahmin edici API 109, 110
scikit-learn imputation belgeleri
referans bağlantısı 109
scikit-learn transfromer API 109
puanlama metrikleri
çok sınıflı sınıflandırma için 200
öz-dikkat mekanizması 544
temel form 545-549
parametrelendirme 549-552
kendi kendine gözetimli öğrenme 561
graf konvolüsyonel ağlar ile yarı denetimli sınıflandırma
referans bağlantısı 666
duyarlılık analizi görevi
RNN modeli, 521-523 için oluşturma
duyarlılık sınıflandırması 574
IMDb film incelemelerinin
duyarlılığı öngörü 513
katmanlar, cümle kodlaması için gömme 517-519
film inceleme verileri, hazırlama 513-516 RNN modeli, oluşturma 520, 521
sepal genişlik 87
seq2seq görevi 540
sekans modelleme 501
RNN'ler ile 502
sekans modelleme görevleri
çoktan çoğa 502
çoktan bire 502
bire-çok 502
sekanslar
500'ü temsil eden
sıralı geriye doğru seçim (SBS) 129-132, 139
sıralı veri 499
500 modelleme
zaman serisi verilerine karşı 500
sıralı özellik seçimi algoritmaları 129 sigmoid
fonksiyonu 61
Sinyal 454
siluet analizi 314
siluet katsayısı 314
siluet parselleri 306
kümeleme kalitesi, ölçme 314-319
benzerlik fonksiyonu 83
basit doğrusal regresyon
270
tek komut, çoklu veri (SIMD) 29 tek
katmanlı ağ 338
tek katmanlı sinir ağı 337, 338
tek bağlantı 320
sklearn.tree.plot_tree
referans bağlantı 94
gevşek değişkenler
doğrusal olmayan ayrılabilir durumla başa çıkmak için kullanılır 77-79
yumuşak kümeleme 311
yumuşak k-ortalamalar 311
soft-margin sınıflandırma
77 softmax işlevi
çoklu sınıflarda sınıf olasılıklarını tahmin etmek için kullanılır.
sınıflandırma 402
softmax regresyon 60
seyrek dikkat 566
spektral grafik konvolüsyonları 665, 666
spektral ağlar ve yerel olarak bağlı grafikler üzerinde ağlar
referans bağlantısı 666
karesel Öklid uzaklığı 307 yığın
652

istifleme
referanslar 223
topluluklar oluşturmak için kullanılır 223
standardizasyon 44, 120, 121
durum-geçiş olasılığı 679
durum-değer fonksiyonu 682
durum-değer fonksiyonu tahmini
Monte Carlo 688 ile
stokastik gradyan inişi (SGD) 46, 47, 71, 261, 279, 338, 469
durdurma kelimesi kaldırma 257
stop-words 257
stride 456 alt
grupları
bulma, kümeleme ile 8
alt örnekleme 463
sonraki ödüller 680
karesel hataların toplamı (SSE) 308
denetimli veri sıkıştırma
doğrusal diskriminant analizi yoluyla 154
denetimli öğrenme 2, 674
gelecek hakkında tahminlerde bulunmak için kullanılır 3, 4
destek vektör makinesi (SVM) 181
lojistik regresyon 79
maksimum marj sınıflandırması 76
destek vektörleri 76
simetrik normalleştirilmiş grafik Laplacian
666 sentezlenmiş görüntü
kalite, konvolüsyonel 612 ile iyileştirme
kalite, Wasserstein GAN 612 ile iyileştirme
Sentetik Azınlık Aşırı Örnekleme Tekniği (SMOTE) 203

T

tablosal veri
eksik değerler, 106'da tanımlama
hedef 12
hedef
değerler
zararın hesaplanması için belirleme 708, 709
t-dağıtılmış stokastik komşu gömme (t-SNE) 163
veri görselleştirme 165-168
TD öğrenme 689
TD tahmini 689, 690
zamansal fark (TD) 678, 684 TensorBoard
445'in kurulması
model, değerlendirme 445-449
tensör nesneleri, PyTorch
model parametreleri, 412-414 ile saklama
model parametreleri, 412 ile güncelleme
tensör işleme birimleri (TPU'lar) 597
tensörler
birleştirme 376, 377
oluşturma, PyTorch'ta 373
veri türü, manipüle etme 374
matematiksel işlemler, 375'e uygulama, 376
şekil, manipüle etme 374
bölme 376, 377
istifleme 376, 377
terim sıklığı-ters belge sıklığı
kelime alaka düzeyi, değerlendirme 252-254
Metin
GPT-2 ile üretilmesi 566-568
metin sınıflandırması
tekrarlayan sinir ağları ile 261
metin verileri
temizlik 254-256
metin belgeleri
LDA 264 ile ayrıştırma
metin işleme
IMDb film inceleme verileri, 247 için hazırlık, 248
zaman serisi verileri 500
sıralı verilere karşı 500

tokenizers paketi
referans bağlantısı 577
belirteçler
belgeler, 256'ya işleme, 257
konu modelleme 264
LDA 264 ile
top-k havuzlama 668
torch.bucketize işlevi
referans bağlantı 433
TorchDrug kütüphanesi 665
referans bağlantısı 665
torch.nn modülü
CNN'i uygulamak için kullanılır 476
toplam varyans 145
Seyrek Hiyerarşik Grafik Sınıflandırıcılara Doğru
referans bağlantısı 668
geleneksel RNN kodlayıcı-kod çözücü mimarisi
seq2seq modelleme görevi için 540
EĞİTİM 11
eğitim örneği 11
eksik değerlerle eleme 107, 108
transformatör
Trainer API 582-586 **transformatör mimarisi**
539, 552 transformatör modelleri ile ince ayar
ince ayar 561, 562
ön eğitim 561, 562
transformatörler 539,
545 transformatörler
paket referans
bağlantısı 579
transformers sürüm 4.9.1
yükleme 567
aktarılmış konvolüsyon 612, 613
Parzen Tahmin Ediciler Ağacı (TPE) yöntemi
191 gerçek sınıf etiketi 23
gerçek negatif (TN) 194
gerçek pozitif oranı (TPR)
195 gerçek pozitif (TP) 194
Zaman boyunca kesilmiş geri yayılım (TBPTT) 510

U

eksik otomatik kodlayıcı 591
yetersiz uyum 73
yönlendirilmemiş çizge 638, 639
tekdüze manifold yaklaşımı ve projeksiyonu (UMAP) 168
referans bağlantisi 168
unigram model 252
birim adım
fonksiyonu 20
etiketsiz veri
büyük ölçekli dil oluşturmak için modeller 561
kaldıraç, GPT ile 563-565
görünmeyen veri örnekleri
14'ü öngörmek
denetimsiz sınıflandırma 8 denetimsiz
boyutsallık azaltma
temel bileşenler analizi yoluyla 139
denetimsiz öğrenme 2, 674
gizli yapıları keşfetmek için kullanılır 7
denetimsiz ön eğitim 561

V

doğrulama eğrileri 180, 183
aşırı uyum, adresleme 185
yetersiz uyum, ele alma 185
doğrulama veri kümesi 132
değer fonksiyonu 594, 682, 683
dinamik programlama ile tahmin etme 686
değer yinelemesi 687
kaybolan gradyan 510

açıklanan varyans oranları 144
varyans azaltma 300 varyasyonel
otomatik kodlayıcı (VAE) 592 venv
referans bağlantısı 17
köşeler 638
VotingClassifier
referans linki 214

W

Wasserstein GAN
sentezlenmiş görüntü kalitesini iyileştirmek için kullanılır 612
Wasserstein GAN (WGAN) 612
zayıf öğrenici
adaptif güçlendirme yoluyla kaldıraçlama 229
ağırlık azalması
L2 düzenlemeye karşı 469
Gradyan cezalı WGAN (WGAN-GP) 628
DCGAN modelini eğitmek için uygulama 629-632
Widrow-Hoff kuralı 36
Şarap veri seti 117
elde etme 117, 142, 226
düğüm içi varyans 300
word2vec modeli 264
kelime büyük harf
kullanımı
255'i yönetmek
kelime alaka düzeyi
terim sıklığı-ters belge sıklığı (tf-idf) aracılığıyla değerlendirme 252-254
kelimeler
özellik vektörlerine dönüştürme 250, 251
kelime köklendirme 256 iş
akışları
boru hatları ile düzene sokulması 171

X

XGBoost
URL 244
243-245 kullanarak
XOR sınıflandırma problemi 419
çözme 419-423

Z

sıfır dolgu 455
sıfır atış görevleri 565, 566

Bu kitabın Ücretsiz PDF kopyasını indirin

Bu kitabı satın aldığınız için teşekkürler!

Hareket halindeyken kitap okumayı seviyorsunuz ancak basılı kitaplarınızı her yere taşıyamıyor musunuz? Satın aldığınız e-Kitap seçtiğiniz cihazla uyumlu değil mi?

Endişelenmeyin, artık her Packt kitabıyla birlikte o kitabın DRM'siz PDF sürümüne hiçbir ücret ödemeden sahip olacaksınız.

Her yerde, her yerde, her cihazda okuyun. Favori teknik kitaplarınızdaki kodları arayın, kopyalayın ve doğrudan uygulamanıza yapıştırın.

Avantajlar bununla da bitmiyor, her gün gelen kutunuzda indirimlere, haber bültenlerine ve harika ücretsiz içeriklere özel erişim elde edebilirsiniz

Avantajlardan yararlanmak için bu basit adımları izleyin:

1. QR kodunu tarayın veya aşağıdaki bağlantıyı ziyaret edin

https://packt.link/free-ebook/9781801819312

2. Satın alma belgenizi gönderin
3. İşte bu kadar! Ücretsiz PDF'inizi ve diğer avantajlarınızı doğrudan e-postanıza göndereceğiz

6587191CA584E
olimpikbarbell@gmail.com