Fəsil 15. RNNs və CNNs vasitəsi ilə ardıcıl verilənlərin emalı

Gələcəyi təxmin etməyi sən bütün zamanlarda edirsən, istər dostunun cümləsini tamamlayarkən, istərsə də səhər yeməyində qəhvə qoxusu təxmin edərkən.Bu fəsildə biz gələcəyi müəyyən mənada proqnozlaşdıra bilən neyron şəbəkə sinifi – rekkurent neyron şəbəkələri və ya təkrarlanan sinir şəbəkəsi (RNN-lər) müzakirə edəcəyik (qismən). RNN-lər zaman sıralı veriləri təhlil etmək qabiliyyətinə malikdir; məsələn, veb saytınızdakı gündəlik aktiv istifadəçi sayı, şəhərinizdə saatlıq temperatur dəyərləri, evinizin gündəlik enerji sərfiyyatı, yaxınlıqdakı avtomobillərin trayektoriyası və daha çoxu. RNN-lər keçmiş nümunələri öyrəndikdən sonra, bu bilikdən istifadə edərək gələcək üçün proqnozlar verə bilir — əlbəttə, əgər keçmişdə müşahidə edilən nümunələr gələcəkdə də eyni qaydada davam edərsə.
Ümumiyyətlə, RNN -lər sabit ölçülü girişlərdən fərqli olaraq, istənilən uzunluqda ardıcıllıqlarla işləyə bilir.Məsələn, cümlələri, mətn sənədlərini və ya audio nümunələrini giriş (input) kimi qəbul edə bilirlər, bu da RNN-ləri avtomatik tərcümə və nitqin mətnə çevrilməsi kimi təbii dil emalı tətbiqləri üçün son dərəcə faydalıdır.
Bu fəsildə əvvəlcə RNN-lərin təməl anlayışlarını və onların zaman üzrə geri yayılma (backpropagation through time) üsulu ilə necə öyrənildiyini nəzərdən keçirəcəyik. Daha sonra isə, RNN-lərdən zaman sırasının proqnozlaşdırılması üçün istifadə edəcəyik. Bununla yanaşı , biz tez-tez zaman sırası proqnozlarında istifadə olunan və RNN-lərlə müqayisə üçün baza model kimi - ARMA (Autoregressive Moving Average) ailəsinə aid modellərə də nəzər salacağıq. Bundan sonra, RNN-lərin üzləşdiyi iki əsas çətinliyi araşdıracağıq:

- Qeyri-sabit qradientlər (11-ci fəsil), hansı ki, təkrarlanan dropout və təkrarlanan layer normalizasiyası kimi müxtəlif texnikalarla azaldıla bilər.

- (Çox) Məhdud qısa müddətli yaddaş, hansı ki, LSTM və GRU hüceyrələri vasitəsilə genişləndirilə bilər.

Ardıcıl verilənləri emal edə bilən yeganə neyron şəbəkə tipi RNN-lər deyil. Kiçik ardıcıllıqlar üçün adi sıx (dense) şəbəkələr də effektiv ola bilər, çox uzun ardıcıllıqlar — məsələn, audio nümunələri və ya mətn üçün isə konvolyusiya neyron şəbəkələri (CNN-lər) kifayət qədər yaxşı nəticə verə bilir. Bu fəsildə bu iki yanaşmanı da müzakirə edəcəyik və fəsli WaveNet -in implementasiyası ilə bitirəcəyik — bu, on minlərlə zaman addımından ibarət ardıcıllıqları emal edə bilən bir CNN arxitekturasıdır.Başlayaq!

Təkrarlanan Neyronlar and Təbəqələr

İndiyə qədər biz yalnız irəli ötürməli (feedforward) neyron şəbəkələrə fokuslanmışdıq. Bu tip şəbəkələrdə aktivasiyalar yalnız bir istiqamətdə — giriş qatından çıxış qatına doğru ötürülür. Təkrarlanan neyron şəbəkə (RNN) isə quruluş etibarilə feedforward şəbəkəyə bənzəsə də, burada əlavə olaraq geri yönəlmiş bağlantılar mövcuddur.

Ən sadə RNN modelinə nəzər salaq: bu model cəmi bir neyronun iştirak etdiyi şəbəkədir. Bu kiçik şəbəkəni zaman oxu boyunca təsvir etmək olar və bu, Şəkil 15-1-in sol hissəsində göstərilib. Hər bir zaman addımında **t** (buna həm də frame deyilir), bu rekurrent neyron həm $\mathbf{x}_{(t)}$ girişlərini, həm də əvvəlki zaman addımındakı öz çıxışını $\hat{\mathbf{y}}_{(t-1)}$ qəbul edir. Burada eyni təkrarlanan neyron hər bir zaman addımında ayrıca təmsil edilir. Bu şəkil RNN-lərin zaman ardıcıllığı boyunca necə işlədiyini daha yaxşı başa düşmək üçün istifadə olunur.

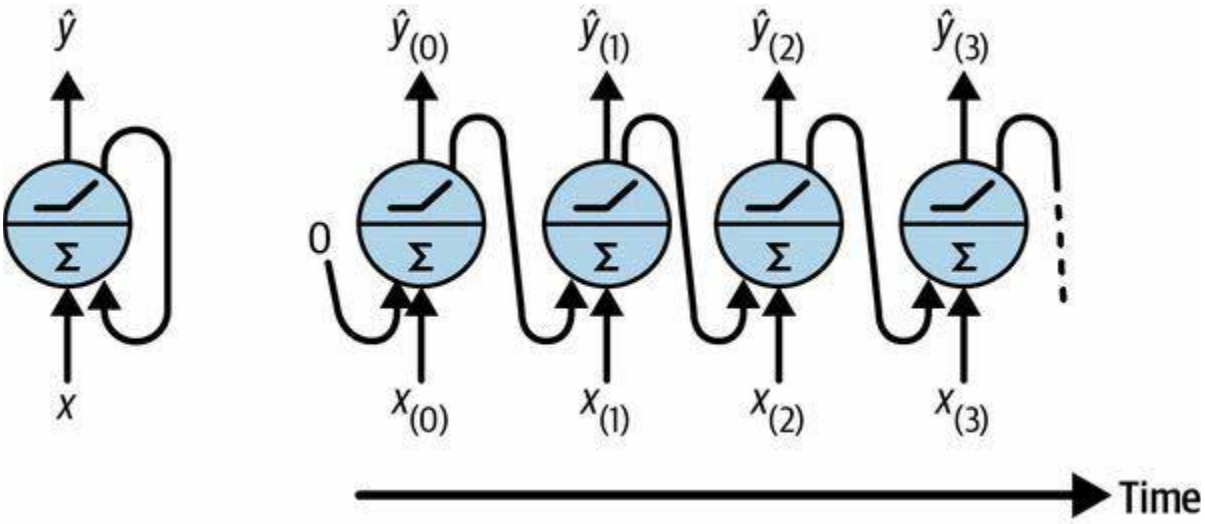


Figure 15 -1. A recurrent neuron(left) unrolled through time (right)

Sən asanlıqla təkrarlanan neyronlar(recurrent neuron) üçün təbəqə yarada bilərsən. Hər bir zaman addımında t, hər bir neyron iki fərqli məlumat qəbul edir. Yəni həm giriş vektoru $\mathbf{x}_{(t)}$ -ni, həm də əvvəlki zaman addımının çıxış vektoru $\hat{\mathbf{y}}_{(t-1)}$ qəbul edir. Bu, Şəkil 15-2-də göstərilmişdir.Qeyd etmək lazımdır ki, həm girişlər, həm də çıxışlar artıq vektor şəklindədir (əgər yalnız bir neyron olsaydı, çıxış(output) sadəcə skalyar olardı).

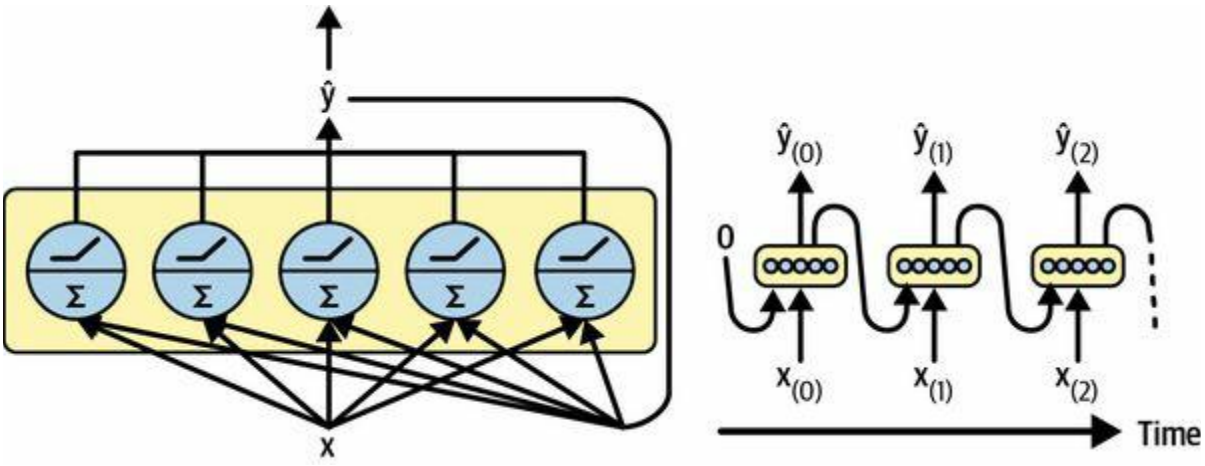


Figure 15-2. A layer of recurrent neurons (left) unrolled through time (right)

Hər bir təkrarlanan neyronun iki dəst çəkisi (weight) var: bunlardan biri girişlər $\mathbf{x_{(t)}}$, digəri isə əvvəlki zaman addımının çıxışları üçün $\hat{\mathbf{y}}_{(t-1)}$. Bu çəkiləri müvafiq olaraq $\mathbf{w_x}$ və $\mathbf{w_{\hat{y}}}$ adlandıraq. Əgər yalnız bir neyron yox, bütöv bir təkrarlanan qat (recurrent layer) nəzərə alınsa, bütün bu çəkiləri iki çəkilər matrisində toplamaq olar: $\mathbf{w_x}$, $\mathbf{w_{\hat{y}}}$.

Beləliklə, bütün təkrarlanan qatın çıxış vektorunu gözlənildiyi kimi hesablamaq olar, bu da tənlik 15-1-də göstərilib. Burada **b** - *bias* (meyl) vektoru, **φ(·)** isə aktivasiya funksiyasıdır (məsələn, ReLU[1]).

Equation 15-1. Output of a recurrent layer for a single instance

$$\hat{y}\,(t) = \phi\; W\; x\; \intercal\; x(t) + W\; \hat{y}\; \intercal\; \hat{y}(t-1) + b$$

Feedforward neyron şəbəkələrində olduğu kimi, təkrarlanan qatın çıxışını da bütün mini-batch üçün bir anda hesablamaq mümkündür. Bunun üçün zaman addımı **t** - dəki bütün girişləri bir giriş matrisi $\mathbf{X_{(t)}}$ - ə yerləşdirmək kifayətdir (bax, Tənlik 15-2).

Equation 15-2. Outputs of a layer of recurrent neurons for all instances in a pass:[mini-batch]

$$\hat{Y}\,(t) = \varphi\; X\,(t)\; W\; x + \hat{Y}\,(t-1)\; W\; \hat{y} + b = \varphi\; X\,(t)\; \hat{Y}\,(t-1)\; W + b \text{ with } W = W\; x$$
$$W\; \hat{y}$$

Bu tənlikdə:

- $\hat{\mathbf{Y}}_{(t)}$ — mini-batch-dəki hər bir nümunə üçün, zaman addımı **t** - də qatın çıxışlarını ehtiva edən **m** × $\mathbf{n}_{\mathbf{neurons}}$ ölçülü matrisdir (m — mini-batch-dəki nümunələrin sayı, $\mathbf{n}_{\mathbf{neurons}}$ — neyronların sayı).
- $\mathbf{X}_{(t)}$ — bütün nümunələrin girişlərini özündə saxlayan **m** × $\mathbf{n}_{\mathbf{inputs}}$ ölçülü matrisdir

  ($\mathbf{n}_{\mathbf{inputs}}$ — giriş xüsusiyyətlərinin sayı).

- $\mathbf{W}_{\mathbf{x}}$ — cari zaman addımındakı girişlər üçün əlaqə çəkilərini özündə saxlayan

  $\mathbf{n}_{\mathbf{inputs}}$ × $\mathbf{n}_{\mathbf{neurons}}$ ölçülü matrisdir.

- $\mathbf{W}_{\acute{\mathbf{Y}}}$ — əvvəlki zaman addımının çıxışları üçün əlaqə çəkilərini (connection weights) özündə saxlayan $\mathbf{n}_{\mathbf{neurons}}$ × $\mathbf{n}_{\mathbf{neurons}}$ ölçülü matrisdir.
- **b** — hər bir neyron üçün bias terminlərini özündə saxlayan ölçüsü $\mathbf{n}_{\mathbf{neurons}}$ olan vektordur.

Çəki matrisləri $\mathbf{W}_{\mathbf{x}}$ və $\mathbf{W}_{\acute{\mathbf{Y}}}$ çox vaxt vertikal istiqamətdə birləşdirilərək ölçüsü

($\mathbf{n}_{\mathbf{inputs}}$ + $\mathbf{n}_{\mathbf{neurons}}$)× $\mathbf{n}_{\mathbf{neurons}}$ olan vahid **W** matrisi formalaşdırılır (bax, Tənlik 15-2-nin ikinci sətrinə).

[$\mathbf{X}_{(t)}$ $\hat{\mathbf{Y}}_{(t-1)}$ **]** yazılışı isə $\mathbf{X}_{(t)}$ və əvvəlki çıxış matrisi $\hat{\mathbf{Y}}_{(t-1)}$ nin horizontal şəkildə birləşdirilməsini göstərir.

Qeyd edək ki, $\hat{Y}_{(t)}$ həm $X_{(t)}$-nin, həm də əvvəlki çıxış olan $\hat{Y}_{(t-1)}$-in funksiyasıdır. Öz növbəsində, $\hat{Y}_{(t-1)}$ isə $X_{(t-1)}$ və ondan əvvəlki çıxış $\hat{Y}_{(t-2)}$-nin funksiyasıdır. Bu zəncirvari əlaqə $X_{(t-2)}$ və $\hat{Y}_{(t-3)}$ kimi əvvəlki addımlara da tətbiq olunur və beləliklə davam edir. Bu səbəbdən, $\hat{Y}_{(t)}$ zaman t = 0 - dan etibarən daxil edilmiş bütün girişlərin funksiyasına çevrilir, (yəni $X_{(0)}$, $X_{(1)}$, ..., $X_{(t)}$ kimi.) zaman t = 0 olduqda isə əvvəlki çıxışlar mövcud olmadığı üçün onlar adətən sıfır kimi qəbul edilir.

Yaddaş Hüceyrələri

Rekurrent neyronun zaman addımı t-dəki çıxışı əvvəlki bütün girişlərin funksiyası olduğu üçün, belə demək olar ki, bu neyronun müəyyən bir yaddaşı mövcuddur. Neyron şəbəkəsində zaman addımları boyunca müəyyən bir vəziyyəti (state) qoruyan hissəyə *yaddaş hüceyrəsi* (memory cell, qısaca cell) deyilir. Tək bir rekurrent neyron və ya rekurrent neyronlardan ibarət bir qat sadə bir yaddaş hüceyrəsi

hesab olunur və bu tip hüceyrələr əsasən qısa ardıcıllıqları öyrənə bilir (adətən təxminən 10 addımlıq, lakin bu limit tapşırıqdan asılı olaraq dəyişə bilər).

Bu fəsildə daha sonra daha mürəkkəb və güclü yaddaş hüceyrəsi növləri ilə tanış olacağıq; onlar daha uzun ardıcıllıqları öyrənmək qabiliyyətinə malikdirlər (təxminən 10 dəfə daha uzun, lakin bu da tapşırıqdan asılıdır).

Hüceyrənin zaman addımı t-dəki vəziyyəti $\mathbf{h}_{(t)}$ ilə işarə edilir (burada *h* "gizli vəziyyət" - *hidden state* mənasını verir) və bu vəziyyət cari girişlər və əvvəlki addımdakı vəziyyətin funksiyasıdır:

$$\mathbf{h}_{(t)} = \mathbf{f}(\mathbf{x}_{(t)}, \mathbf{h}_{(t-1)})$$

Eyni zamanda hüceyrənin zaman t-dəki çıxışı $\hat{\mathbf{y}}_{(t)}$ ilə göstərilir və o da əvvəlki vəziyyət və cari girişlərə bağlıdır. İndiyə qədər müzakirə etdiyimiz sadə hüceyrələrdə çıxış vəziyyətinə tam bərabərdir. Lakin daha kompleks hüceyrələrdə bu belə olmur; bu fərq **15.3-cü** şəkildə təsvir edilmişdir.

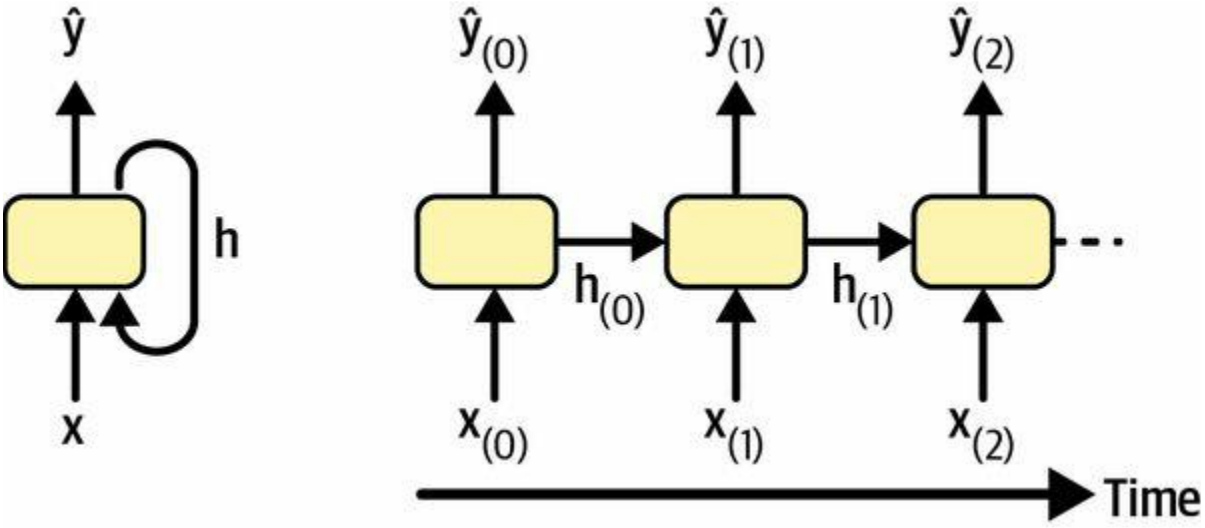


Figure 15-3. A cell's hidden state and its output may be different

Giriş və Çıxış Ardıcıllıqları

RNN eyni anda həm bir giriş ardıcıllığını qəbul edə, həm də ona qarşılıq bir çıxış ardıcıllığı istehsal edə bilər (Təsvir 15-4-ün sol yuxarı küncündəki şəbəkəyə baxın). Bu cür *ardıcıllıqdan-ardıcıllığa (sequence-to-sequence)* şəbəkələr zaman seriyalarının proqnozlaşdırılması üçün faydalıdır. Məsələn, evinizin gündəlik enerji sərfiyyatını öncədən təxmin etmək üçün istifadə edilə bilər: şəbəkəyə son N günün məlumatlarını ötürürsünüz və onu bir gün irəliyə aid enerji sərfiyyatını çıxış kimi verməyə öyrədirsiniz, yəni N – 1 gün əvvəldən sabaha qədər olan məlumat proqnozlaşdırılır.

Alternativ olaraq, şəbəkəyə bir giriş ardıcıllığı verib yalnız son çıxışı nəzərə ala bilərsiniz (Təsvir 15-4-ün sağ yuxarı küncündəki şəbəkəyə baxın).Bu tip quruluş ardıcıllıqdan-vektora (sequence-to-vector) şəbəkə adlanır.

Məsələn, şəbəkəyə bir film rəyi ilə uyğun gələn sözlər ardıcıllığını daxil edə bilərsiniz və şəbəkə həmin rəylə bağlı emosional qiymət (sentiment score) çıxışı verər. Bu çıxış məsələn, **0** (nifrət) ilə **1** (sevgi) arasında dəyişən bir qiymət ola bilər.

Əksinə, şəbəkəyə hər zaman addımında eyni giriş vektorunu təkrar-təkrar verə bilərsiniz və bu zaman şəbəkə bir çıxış ardıcıllığı istehsal edər (Şəkil 15-4-ün aşağı sol küncündəki şəbəkəyə baxın). Bu cür model vektordan-ardıcıllığa (vector-to-sequence) şəbəkə adlanır.
Məsələn, giriş bir şəkil və ya Konvolyusiya Neyron Şəbəkəsinin (CNN) çıxışı ola bilər və çıxış isə həmin şəkil üçün yaradılan şərh və ya təsvir (caption) ola bilər.

Nəhayət, bir ardıcıllıqdan-vektora şəbəkə — yəni enkoder (encoder) — qurmaq, daha sonra isə onu vektordan-ardıcıllığa şəbəkə — dekoder (decoder) — ilə izləmək mümkündür.
(Şəkil,15-4 -ün sağ aşağıdakı şəbəkəyə baxın)

Bu yanaşma məsələn, bir cümləni bir dildən digərinə tərcümə etmək üçün istifadə oluna bilər. Şəbəkəyə müəyyən bir dildə cümlə daxil edilir, enkoder həmin cümləni tək bir vektor təsvirinə çevirir, sonra isə dekoder bu vektoru digər dildəki cümləyə dekodlaşdırır (yəni yenidən tərcümə edir).

Bu iki mərhələli model enkoder–dekoder arxitekturası adlanır və sadəcə birbaşa ardıcıllıqdan-ardıcıllığa RNN ilə tərcümə etməkdən daha yaxşı nəticə verir (Şəkil 15-4-dəki yuxarı sol model kimi). Çünki cümlənin son sözləri tərcümənin ilk sözlərinə təsir göstərə bilər; buna görə də, tərcüməyə başlamazdan əvvəl bütün cümləni görmək və onu tam işləmək lazımdır.

Bu arxitekturanın implementasiyası ilə 16-cı fəsildə daha ətraflı tanış olacağıq (orada görəcəyik ki, əslində bu model Şəkil 15-4-ün təklif etdiyindən bir qədər daha mürəkkəbdir).

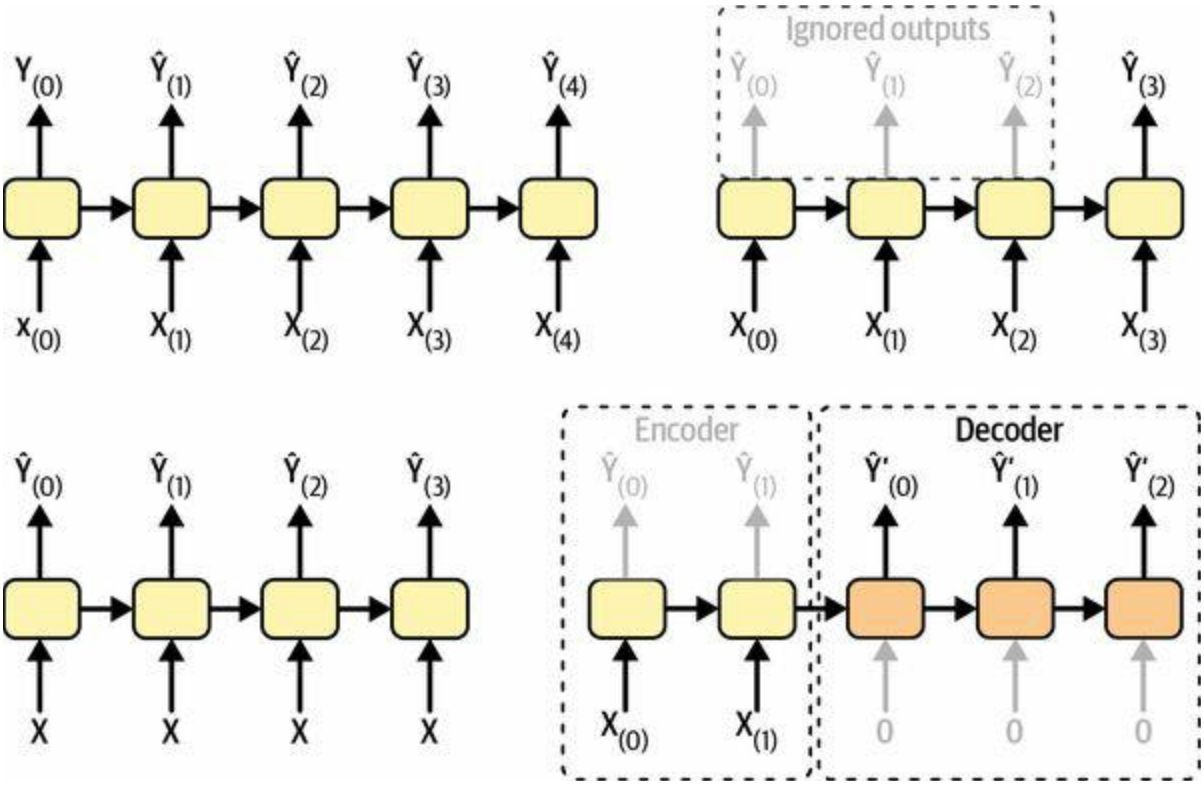


Figure 15-4. Sequence-to-sequence (top left), sequence-to-vector (top right), vector-to-sequence (bottom left), and encoder–decoder (bottom right) networks

Bu çevik yanaşma ümidverici səslənir, bəs rekurrent neyron şəbəkəni necə öyrətmək (train) olar?

RNN-lərin Öyrədilməsi

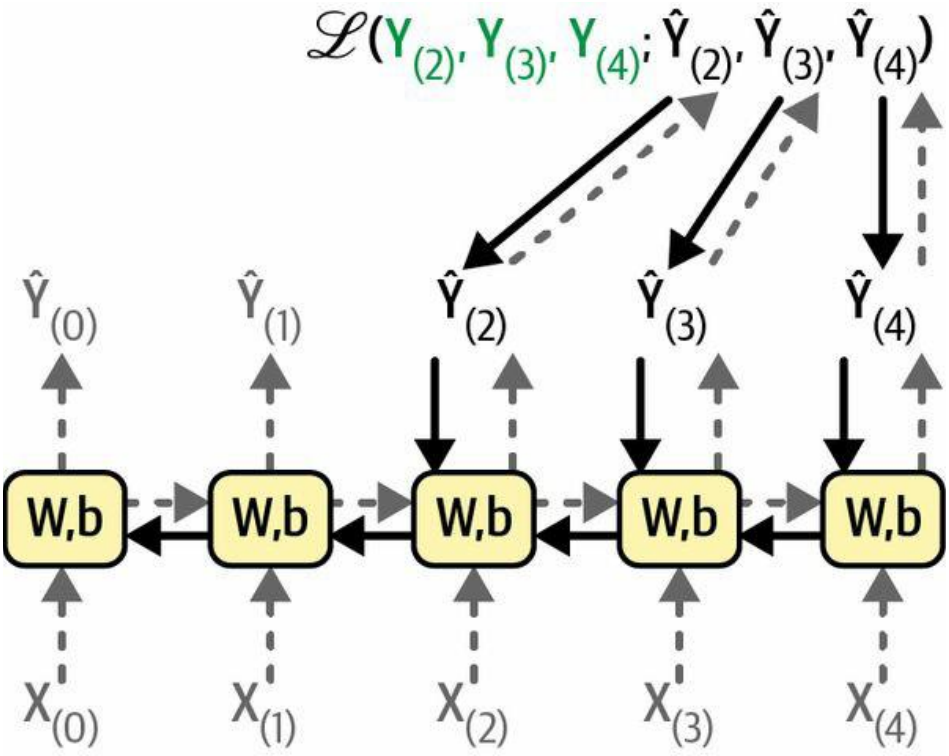
RNN-i öyrətmək üçün əsas üsul onu zaman boyunca unroll (yenidən açmaq) (bir qədər əvvəl etdiyimiz kimi) və sonra adi geri yayılma (backpropagation) üsulundan istifadə etməkdir.
(Təsvir 15-5 baxın)

Bu strategiya zaman üzrə geri yayılma (Backpropagation Through Time - BPTT) adlanır.

Adi geri yayılmada (backpropagation) olduğu kimi, burada da əvvəlcə açılmış (unrolled) şəbəkə üzrə irəli keçid mərhələsi həyata keçirilir (şəkildə kəsik oxlarla göstərilmişdir). Daha sonra çıxış ardıcıllığı itki funksiyası $\mathscr{L}(Y_1, Y_2, \ldots, Y_t; \hat{Y}_1, \hat{Y}_2, \ldots, \hat{Y}_t)$ ilə qiymətləndirilir (burada Y_i hədəfi, $\hat{Y}_i$ isə müvafiq proqnozu, T isə maksimal zaman addımını ifadə edir). Qeyd etmək lazımdır ki, bu itki funksiyası bəzi çıxışları nəzərə almaya bilər. Vektora xəritələnən ardıcıllıq modelində (sequence-to-vector RNN) giriş kimi verilən bütün ardıcıllıq boyunca şəbəkə bir çox çıxış yaradır. Lakin bu çıxışlardan yalnız ən sonuncusu nəzərə alınır, digərləri isə hesablamalarda istifadə olunmur və sadəcə ötürülür. Çünki bu tip modellərdə məqsəd bütün ardıcıllıq üzrə məlumatı toplayıb yekun olaraq yalnız bir nəticə, yəni bir vektor əldə etməkdir. Məsələn, bir cümləni təhlil edib onun emosional tonunu (müsbət/ mənfi) müəyyən etmək kimi — burada yalnız son çıxışdan istifadə edilir, çünki yekun qərar üçün o kifayətdir.

Şəkil 15-5-də isə itki funksiyası yalnız son üç çıxış əsasında hesablanır.

Sonra isə həmin itki funksiyasının qradientləri şəbəkə üzrə geriyə doğru yayılır (şəkildə düz oxlarla göstərilmişdir). Bu nümunədə $\hat{Y}_1$ və $\hat{Y}_2$ çıxışları itki hesablanmasında istifadə edilmədiyi üçün, qradientlər bu çıxışlar üzərindən geriyə axmır; onlar yalnız $\hat{Y}_3$, $\hat{Y}_4$ və $\hat{Y}_5$ vasitəsilə ötürülür. Üstəlik, hər zaman addımında eyni parametrlər (W və b) istifadə edildiyindən, geri yayılma prosesi zamanı bu parametrlərin qradientləri dəfələrlə yenilənir. Geri yayılmanın bu mərhələsi başa çatdıqdan və bütün qradientlər hesablandıqdan sonra, zaman üzrə geri yayılma (Backpropagation Through Time — BPTT) metodu qradient enişi (gradient descent) addımı yerinə yetirərək parametrləri yeniləyir.Bu, klassik geri yayılma prosesindən fərqlənmir.



Təsvir 15-5. Backpropagation through time

Xoşbəxtlikdən Keras bu qarışıqlığın qarşısını alır, gördüyün kimi. Amma ora keçməzdən əvvəl, gəlin bir zaman seriyası yükləyək və onu klassik alətlərdən istifadə edərək təhlil etməyə başlayaq. Bu, qarşımızdakı məlumatları daha yaxşı anlamağa, ilkin əsas göstəriciləri (baseline metrics) əldə etməyə kömək edəcək.

Zaman seriyasının proqnozlaşdırılması

Yaxşı! Təsəvvür edin ki, sizi Çikaqo Nəqliyyat İdarəsinə data scientist kimi işə götürüblər. İlk tapşırığınız ertəsi gün avtobus və metrodan istifadə edəcək sərnişinlərin sayını proqnozlaşdırmaq üçün bir model qurmaqdır. Bunun üçün sizin 2001-ci ildən bəri gündəlik sərnişin istifadə məlumatlarına çıxışınız var. Gəlin bu prosesi birlikdə addım-addım həyata keçirək. Əvvəlcə məlumatları yükləyib təmizləyəcəyik:

```
import pandas as pd
from pathlib import Path

path = Path("datasets/ridership/CTA_-_Ridership_-_Daily_Boarding_Totals.csv")
df = pd.read_csv(path, parse_dates=["service_date"])
df.columns = ["date", "day_type", "bus", "rail", "total"]  # shorter names
df = df.sort_values("date").set_index("date")
df = df.drop("total", axis=1)  # no need for total, it's just bus + rail
df = df.drop_duplicates()  # remove duplicated months (2011-10 and 2014-07)
```

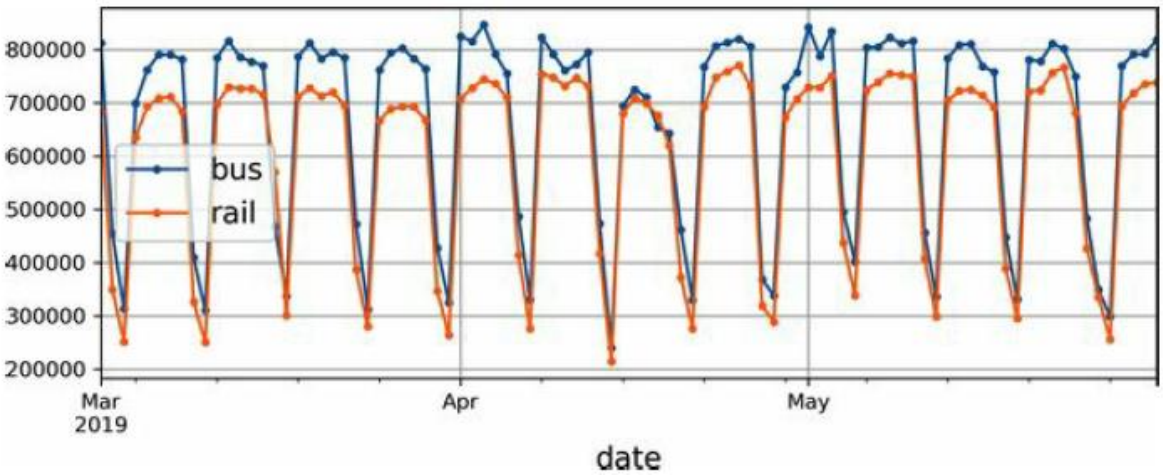
CSV faylını yükləyirik, sütun adlarını qısaldırıq, sətrləri tarixə görə sıralayırıq, artıq olan "total" sütununu silirik və təkrarlanan sətrləri çıxarırıq. İndi isə ilk bir neçə sətrin necə göründüyünə baxaq:

```
>>> df.head()
           day_type    bus    rail
date
2001-01-01        U  297192  126455
2001-01-02        W  780827  501952
2001-01-03        W  824923  536432
2001-01-04        W  870021  550011
2001-01-05        W  890426  557917
```

2001-ci il yanvarın 1-də Çikaqoda 297,192 nəfər avtobusa, 126,455 nəfər isə metroya minmişdir. day_type sütunu isə günün növünü göstərir: burada **W** həftə günləri (weekdays), **A** şənbə günlərini (Saturdays), **U** isə bazar günləri və ya bayram günlərini (Sundays or holidays) ifadə edir.

İndi isə 2019-cu ildə bir neçə ay üzrə avtobus və metro istifadəsinin göstəricilərini vizuallaşdıraraq, məlumatların necə göründüyünü analiz edək (bax: Təsvir 15-6).

```
import matplotlib.pyplot as plt

df["2019-03":"2019-05"].plot(grid=True, marker=".", figsize=(8, 3.5))
plt.show()
```



Təsvir 15-6. Daily ridership in Chicago

Qeyd etmək lazımdır ki, Pandas kitabxanası tarix aralığı təyin edərkən həm başlanğıc, həm də son ayı daxil edir. Bu səbəbdən, göstərilən qrafikdə məlumatlar 1 martdan başlayaraq 31 maya qədər olan dövrü əhatə edir. Bu cür məlumatlar zaman seriyası (time series) adlanır — yəni fərqli zaman addımlarında müşahidə olunan dəyərlərdən ibarət verilənlər, adətən bərabər aralıqlarla qeydə alınır.

Bundan daha dəqiq desək, burada hər bir zaman addımında bir neçə dəyər qeyd olunduğu üçün bu tip məlumat çoxdəyişənli zaman seriyası (multivariate time series) adlanır. Əgər biz yalnız bus (avtobus) sütununa baxsaydıq, bu zaman bir dəyişənli zaman seriyası (univariate time series) əldə etmiş olardıq, çünki hər zaman addımında yalnız bir dəyər mövcud olardı.

Zaman seriyası ilə işləyərkən ən əsas məqsəd gələcək dəyərləri proqnozlaşdırmaqdır (forecasting). Bu fəsildə də məhz bu məsələyə diqqət yetiriləcəkdir. Bununla yanaşı, digər tətbiqlərə keçmişdəki çatışmayan dəyərlərin tamamlanması (imputation), sinifləndirmə (classification), anomaliya aşkarlanması (anomaly detection) və digər tapşırıqlar daxildir.

Təsvir 15-6-ya nəzər yetirdikdə aydın görünür ki, hər həftə təkrar olunan oxşar bir nümunə mövcuddur. Bu, həftəlik mövsümlük (weekly seasonality) adlanır. Hətta bu nümunə o qədər güclüdür ki, sadəcə olaraq bir həftə əvvəlki dəyərləri təkrarlamaqla sabahkı sərnişin sayını proqnozlaşdırmaq mümkündür və bu yanaşma kifayət qədər yaxşı nəticə verə bilər. Bu üsul naïv proqnozlaşdırma (naive forecasting) adlanır — yəni proqnoz vermək üçün sadəcə keçmişdəki bir dəyəri təkrarlamaq.

Naïv proqnozlaşdırma tez-tez əsas müqayisə göstəricisi (baseline) kimi istifadə olunur və bəzi hallarda onu üstələmək belə çətin ola bilər.

QEYD

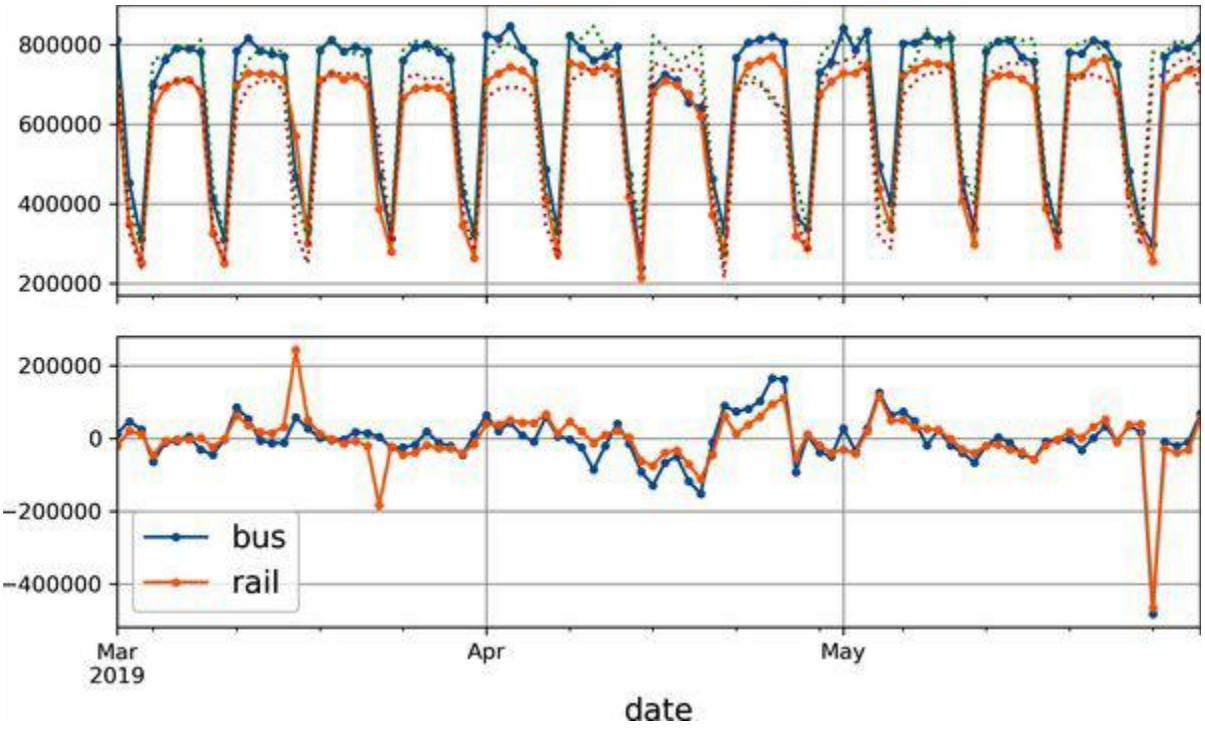
Ümumiyyətlə, naïv proqnozlaşdırma dedikdə, son məlum olan dəyəri təkrarlamaq nəzərdə tutulur (məsələn, sabahın bu günlə eyni olacağını proqnozlaşdırmaq kimi). Lakin bizim nümunədə güclü həftəlik mövsümlük təsir olduğuna görə, bir həftə əvvəlki dəyəri təkrarlamaq daha yaxşı nəticə verir.

Bu naïv proqnozları vizuallaşdırmaq üçün həm avtobus, həm də metro üçün zaman seriyalarını qrafikdə göstərək və eyni zamanda həmin seriyaların bir həftə gecikdirilmiş (yəni sağa doğru sürüşdürülmüş) variantlarını da nöqtəli xətlərlə əlavə edək. Bundan əlavə, bu iki seriya arasındakı fərqi də qrafikdə göstərəcəyik — yəni **t** zamanındakı dəyərdən **t – 7** zamanındakı dəyəri çıxacağıq; bu əməliyyat fərqləndirmə adlanır (bax: Təsvir 15-7).

```
diff_7 = df[["bus", "rail"]].diff(7)["2019-03":"2019-05"]

fig, axs = plt.subplots(2, 1, sharex=True, figsize=(8, 5))
df.plot(ax=axs[0], legend=False, marker=".")  # original time series
df.shift(7).plot(ax=axs[0], grid=True, legend=False, linestyle=":")  # lagged
diff_7.plot(ax=axs[1], grid=True, marker=".")  # 7-day difference time series
plt.show()
```

Heç də pis deyil! Fikir verin ki, gecikdirilmiş zaman seriyası (lagged time series) orijinal zaman seriyasını necə yaxından izləyir. Əgər zaman seriyası özünün gecikdirilmiş versiyası ilə əlaqəlidirsə, buna avtokorrelyasiya (autocorrelation) deyilir. Göründüyü kimi, fərq kifayət qədər kiçikdir, yalnız may ayının sonunda fərq daha böyükdür. Bəlkə həmin dövrdə bir bayram günü olub? Gəlin **day_type** sütununu yoxlayaq:

```
>>> list(df.loc["2019-05-25":"2019-05-27"]["day_type"])
['A', 'U', 'U']
```



Təsvir 15-7. Time series overlaid with 7-day lagged time series (top), and difference between t and t –7 (bottom)

Doğrudan da, həmin dövrdə uzun bir həftəsonu olub: bazar ertəsi Memorial Day (Xatirə Günü) bayramı imiş. Əslində bu sütundan istifadə etməklə proqnozlarımızı daha da yaxşılaşdıra bilərik. Lakin indi sadəcə olaraq 2019-cu ilin mart, aprel və may aylarını əhatə edən bu üç aylıq dövr üzrə orta mütləq xətanı (mean absolute error — MAE) hesablayaq ki, təxmini bir göstəricimiz olsun:

```
>>> diff_7.abs().mean()
bus     43915.608696
rail    42143.271739
dtype: float64
```

Naïv proqnozlarımız üçün orta mütləq xəta (MAE) avtobus sərnişinləri üçün təxminən 43,916, metro sərnişinləri üçün isə 42,143 nəfər təşkil edir.Bu dəyərlərin nə qədər yaxşı və ya pis olduğunu bir baxışda müəyyən etmək çətindir. Buna görə də, proqnoz səhvlərinin nisbətən nə qədər böyük olduğunu anlamaq üçün onları hədəf dəyərlərə bölək.Bu yanaşma, səhvləri nisbi ölçüdə qiymətləndirməyə imkan verir.

```
>>> targets = df[["bus", "rail"]]["2019-03":"2019-05"]
>>> (diff_7 / targets).abs().mean()
bus     0.082938
rail    0.089948
dtype: float64
```

İndi hesabladığımız göstərici Orta Mütləq Faiz Xətasıdır (Mean Absolute Percentage Error - MAPE). Görünür ki, sadə proqnozlarımız avtobus nəqliyyatı üçün təxminən 8.3%, dəmir yolu nəqliyyatı üçün isə təxminən 9.0% MAPE nəticəsi verir.

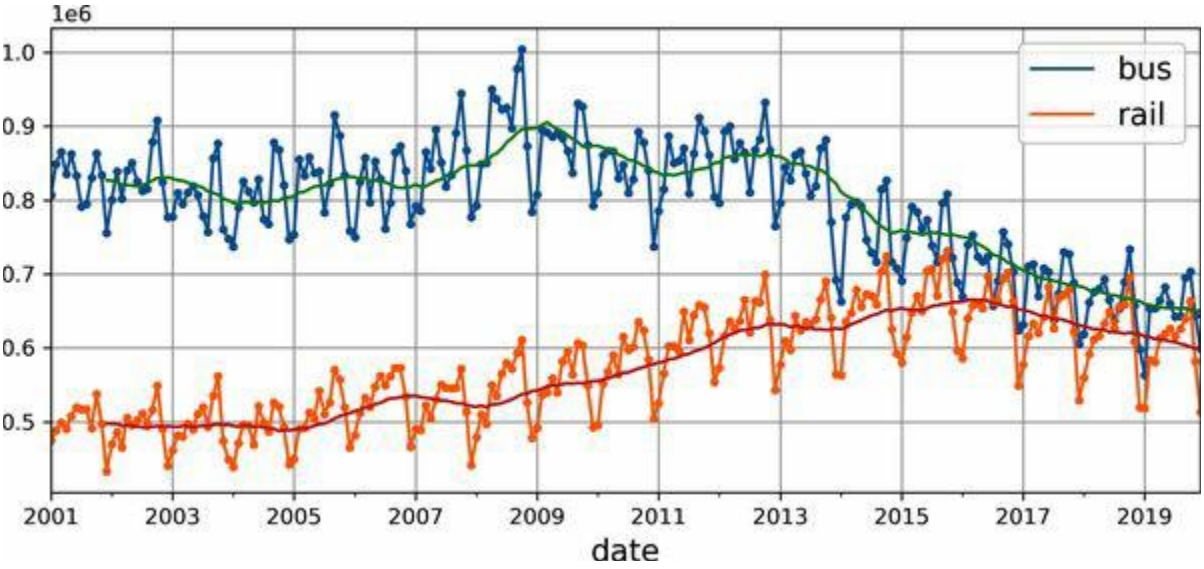
Diqqətçəkən məqam odur ki, dəmiryolu proqnozları üçün orta mütləq xəta (Mean Absolute Error - MAE) göstəricisi avtobus proqnozlarındakından bir qədər yaxşı görünür. Amma MAPE göstəricisində isə əks nəticə müşahidə olunur. Bunun səbəbi budur ki, avtobusla daşınan sərnişinlərin sayı dəmir yolu ilə müqayisədə daha çoxdur, nəticədə proqnoz xətalarının (səhvlərinin) dəyəri də təbii olaraq daha yüksək çıxır. Lakin xətaları nəticənin böyüklüyü ilə müqayisə etdikdə (yəni faiz nisbətində ölçdükdə) görünür ki, əslində avtobus üzrə proqnozlar dəmir yolu üzrə proqnozlardan bir qədər dəqiqdir.

İpucu

Orta Mütləq Səhv (MAE), Orta Mütləq Faiz Səhvi (MAPE) və Orta Kvadrat Səhv (MSE) proqnozların dəqiqliyini qiymətləndirmək üçün ən çox istifadə olunan göstəricilərdəndir. Lakin düzgün göstəricini(metric) seçmək, həmişə olduğu kimi, tapşırıqdan aslıdır. Məsələn, əgər layihənizdə böyük error -lar kiçik error-lardan kvadrat olaraq daha çox ziyan verirsə, o zaman MSE məqsədəuyğun ola bilər, çünki o, böyük səhvləri daha sərt penalize(cəzalandırır) edir.

Zaman seriyasına baxanda, əhəmiyyətli aylıq mövsümlük (seasonality) görünmür, amma illik mövsümlüyün olub-olmadığını yoxlayaq. Biz 2001–2019-cu illər arasındakı məlumatlara baxacağıq. Məlumatların təhlilində "data snooping" riskini azaltmaq üçün daha sonrakı dövrlərin məlumatlarını hazırda nəzərə almayacağıq. Həmçinin, uzunmüddətli tendensiyaları viziuallaşdırmaq üçün hər bir seriya üzrə 12 aylıq sürüşən orta hesabı da qrafikdə göstərəcəyik (bax: Təsvir 15-8).

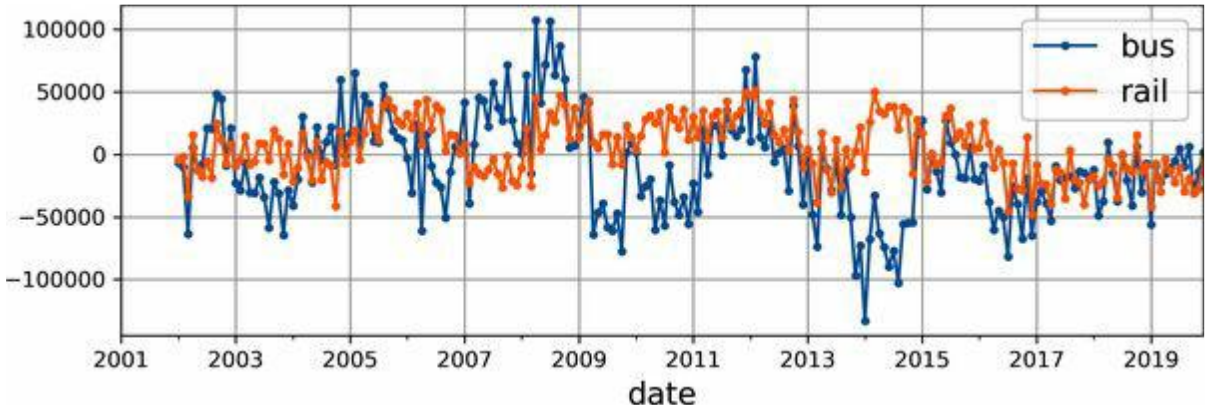
```
period = slice("2001", "2019")
df_monthly = df.resample('M').mean()  # compute the mean for each month
rolling_average_12_months = df_monthly[period].rolling(window=12).mean()

fig, ax = plt.subplots(figsize=(8, 4))
df_monthly[period].plot(ax=ax, marker=".")
rolling_average_12_months.plot(ax=ax, grid=True, legend=False)
plt.show()
```



Təsvir 15-8. İllik mövsümlük və uzunmüddətli tendensiyalar

Bəli, illik mövsümlüyün mövcudluğu aydın şəkildə görünür, baxmayaraq ki, bu mövsümlük həftəlik mövsümlüyə nisbətən daha çox şaxələnmə (noisy) ehtiva edir və qatar seriyası avtobus seriyasından daha nəzərəçarpandır. Hər il təxminən eyni tarixlərdə pik və çökəkliklərin müşahidə olunması bunu təsdiqləyir. İndi isə 12 aylıq fərqin viziuallaşdırılması nəticələrinə baxaq (bax: Təsvir 15-9):

```
df_monthly.diff(12)[period].plot(grid=True, marker=".", figsize=(8, 3))
plt.show()
```



Təsvir 15-9. 12 aylıq fərq

Diqqət yetirin ki, fərqləndirmə (differencing) təkcə illik mövsümlüyü aradan qaldırmaqla kifayətlənmir, həm də uzunmüddətli tendensiyaları da aradan qaldırır.Məsələn, 2016-cı ildən 2019-cu ilə qədər zaman

seriyasında müşahidə olunan xətti eniş tendensiyası zaman seriyalarını fərqləndirməklə təxminən sabit mənfi dəyərə çevrilir.

Əslində, fərqləndirmə zaman seriyasından həm trendi, həm də mövsümlüyü aradan qaldırmaq üçün geniş istifadə olunan standart texnikadır. Statik zaman seriyası üzərində işləmək daha asandır; belə bir zaman seriyasının statistik xüsusiyyətləri zamanla dəyişmir və heç bir mövsümlük və ya trend ehtiva etmir. Fərqləndirilmiş zaman seriyası üzərində düzgün proqnozlar verə bildikdən sonra, əvvəlcədən çıxardığınız keçmiş dəyərləri əlavə etməklə, bu proqnozları orijinal zaman seriyası üçün asanlıqla hesablamaq mümkündür.

Bəlkə də düşünürsünüz ki, biz yalnız sabahkı sərnişindaşımaları proqnozlaşdırırıq və buna görə də uzunmüddətli naxışlar qısa müddətli naxışlardan daha az əhəmiyyət kəsb edir. Bu fikir doğrudur, lakin uzunmüddətli tendensiyaları nəzərə almaqla proqnozların dəqiqliyini bir qədər yaxşılaşdırmaq mümkündür.Məsələn, 2017-ci ilin oktyabr ayında gündəlik avtobus sərnişindaşımaları təxminən 2,500 nəfər azalmışdı ki, bu da həftə ərzində təxminən 570 nəfər az sərnişin deməkdir. Belə ki, əgər 2017-ci ilin oktyabr ayının sonundayıqsa, sabahkı sərnişindaşımaları üçün proqnozu keçən həftənin müvafiq dəyərindən 570-ni çıxmaqla vermək məntiqli olar. Trendin nəzərə alınması proqnozlarınızın orta hesabla dəqiqliyini bir qədər artıracaq.

İndi sərnişindaşımaların zaman seriyası analizinin əsas anlayışları — mövsümlük, trend, fərqləndirmə və sürüşən ortalamalarla tanış olduğunuz üçün, gəlin vaxt seriyalarının təhlilində geniş istifadə olunan və çox məşhur olan statistik model ailəsinə qısa nəzər salaq.

ARMA model ailəsi

1930- cu ildə Herman Wold tərəfindən yaradılan avtoreqressiv hərəkətli orta modeldən (ARMA) danışacağıq. ARMA modeli proqnozlarını keçmiş müşahidələrin çəki ilə vurulmuş cəminə əsaslanaraq hesablayır və bu proqnozları hərəkətli orta komponenti əlavə etməklə düzəldir – bu, daha əvvəl müzakirə etdiyimiz yanaşmaya çox bənzəyir.

Xüsusilə, hərəkətli orta komponent son bir neçə proqnoz xətasının (yəni, əvvəlki proqnozların real dəyərlərlə fərqinin) çəki ilə vurulmuş cəmi əsasında hesablanır. Bu yanaşma modelin yalnız keçmiş müşahidələrə deyil, həm də keçmiş səhvlərə reaksiya verə bilməsinə imkan yaradır.

Bu modelin proqnozları necə hesabladığını isə Tənlik 15-3 göstərir.

Tənlik 15-3. ARMA modelindən istifadə edərək proqnozlaşdırma

$$y^{(t)}=\Sigma i=1p\alpha i\ y(t-i)+\Sigma i=1q\theta i\ \epsilon(t-i) \text{with } \epsilon(t)=y(t)-y^{(t)}$$

Bu tənlikdə:

- $\hat{\mathbf{y}}_{(t)}$ – modelin zaman addımı t üçün proqnozudur.
- $\mathbf{y}_{(t)}$ – zaman addımında t zaman seriyasının dəyəridir.
- Birinci cəm – zaman sırasının əvvəlki p dəyərinin çəki ilə vurulmuş cəmidir. Burada çəkilər α (alfa) ilə göstərilir və model tərəfindən öyrənilir. p ədədi hiperparametrdir, yəni model qurularkən təyin edilir və modelin keçmişə nə qədər uzağa baxacağını müəyyənləşdirir. Bu cəm modelin autoregressiv (AR) komponentini təşkil edir: keçmiş dəyərlərə əsaslanaraq reqressiya aparır.
- İkinci cəm – əvvəlki q proqnoz səhvlərinin çəki ilə vurulmuş cəminidir. Buradakı $\boldsymbol{\varepsilon}_{(t)}$ ağırlıqlı çəkilər $\boldsymbol{\theta}_{(t)}$ (theta) ilə göstərilir və model tərəfindən öyrənilir. q hiperparametrdir və bu, modelin neçə addım geriyə – keçmişdəki səhvlərə nəzər salacağını müəyyənləşdirir. Bu cəm modelin hərəkətli orta (MA) komponentini təşkil edir.

Ən önəmlisi, bu model zaman sırasının stasionar olduğunu fərz edir. Yəni statistik xüsusiyyətləri (ortalama, dispersiya və s.) zamanla dəyişməməlidir. Əgər zaman sırası stasionar deyilsə, o zaman fərqləndirmə (differencing) tətbiq etmək kömək edə bilər.

Fərqləndirmə – zaman sırasındakı ardıcıl iki dəyərin fərqini almaqla həyata keçirilir. Bu əməliyyat zaman sırasının törəməsinə (dəyişmə sürətinə) yaxın bir təxmini verir: başqa sözlə, hər addımda zaman sırasının meylini (slopunu) göstərir. Beləliklə, xətti trend aradan qaldırılır və nəticədə sabit bir sıra əldə olunur.

Məsələn:
Əgər fərqləndirməni bu zaman sırasına tətbiq etsək: **[3, 5, 7, 9, 11]**
Alınan fərqləndirilmiş sıra belə olar:**[2, 2, 2, 2]**

Əgər orijinal zaman sırası xətti trend (linear trend) əvəzinə kvadratik trendə (quadratic trend) malikdirsə, o zaman bir dəfə fərqləndirmə kifayət etməyəcək.

Məsələn, [1, 4, 9, 16, 25, 36] seriyası bir növbəli fərqdən sonra [3, 5, 7, 9, 11] olur, lakin ikinci tur fərqi həyata keçirsək, [2, 2, 2, 2] əldə edirik. Buna görə də, iki dövrəli fərqin həyata keçirilməsi kvadratik meylləri aradan qaldıracaq. Daha ümumi şəkildə, fərqləndirmənin ardıcıl d raundunun yerinə yetirilməsi zaman seriyasının $\mathbf{d^{th}}$-tərtibli törəməsinin təqribi hesablanmasını hesablayır və beləliklə, d sırasına qədər çoxhədli meylləri aradan qaldırır. Bu hiperparametr, d, inteqrasiya qaydası adlanır.

Fərqləndirmə, 1970-ci ildə Corc Box və Gwilym Jenkins tərəfindən Time Series Analysis (Wiley) kitabında təqdim edilmiş avtoreqressiv inteqrasiya olunmuş hərəkətli ortalama (ARIMA) modelinin əsas töhfəsidir: bu model vaxt seriyasını daha stasionar etmək üçün müxtəlif dövrlərdə fərqləndirir, ardınca isə

müntəzəm ARMA modeli. Bu ARMA modeli proqnozlar verərkən istifadə olunur və fərqlə çıxarılan terminlər daha sonra yenidən daxil edilir.

ARMA ailəsinin son üzvü mövsümi ARIMA (SARIMA) modelidir: bu model zaman seriyasını ARIMA ilə eyni şəkildə modelləşdirir, həm də eyni ARIMA yanaşmasından istifadə edərək müəyyən tezlik (məsələn, həftəlik) üçün mövsümi komponenti modelləşdirir. Onun cəmi yeddi hiperparametri var: ARIMA ilə eyni hiperparametrlər p, d və q, üstəgəl mövsümi modeli modelləşdirmək üçün əlavə P, D və Q hiperparametrləri və nəhayət, s ilə işarələnən mövsümi modelin dövrü. P, D və Q hiperparametrləri p, d və q ilə eynidir, lakin t–s, t–2s, t–3s və s. kimi zaman seriyalarını modelləşdirmək üçün istifadə olunur.

Gəlin görək SARIMA modelini dəmir yolu vaxt seriyasına necə uyğunlaşdıraq və ondan sabahkı sərnişinlərin sayını proqnozlaşdırmaq üçün istifadə edək. Tutaq ki, bu gün 2019-cu ilin may ayının son günüdür və biz 1 iyun 2019-cu il üçün "sabah" üçün dəmir yolu sərnişinlərinin sayını proqnozlaşdırmaq istəyirik.

Bunun üçün bir çox müxtəlif statistik modelləri, o cümlədən ARMA modeli və onun ARIMA sinfi tərəfindən həyata keçirilən variantlarını ehtiva edən *statsmodels* kitabxanasından istifadə edə bilərik:

```
from statsmodels.tsa.arima.model import ARIMA

origin, today = "2019-01-01", "2019-05-31"
rail_series = df.loc[origin:today]["rail"].asfreq("D")



model = ARIMA(rail_series,
              order=(1, 0, 0),
              seasonal_order=(0, 1, 1, 7))
model = model.fit()
y_pred = model.forecast()  # returns 427,758.6
```

Bu kod nümunəsində əvvəlcə:

- ARIMA sinfini (class) statsmodels kitabxanasından idxal edirik. Daha sonra dəmir yolu sərnişin daşımaları ilə bağlı məlumatları 2019-cu ilin əvvəlindən "bugünkü günə" qədər alırıq. asfreq("D") metodu ilə zaman sırasının tezliyini gündəlik (daily) olaraq təyin edirik. Bu konkret halda məlumatlar onsuz da gündəlik olduğu üçün məlumatın özündə heç bir dəyişiklik baş vermir. Lakin bu addım vacibdir, çünki əks halda ARIMA sinfi tezliyi avtomatik müəyyən etməyə çalışacaq və bu da xəbərdarlıq mesajı (warning) ilə nəticələnə bilər.
- Ardınca, "bugünə" qədər olan bütün məlumatları ötürərək bir ARIMA nümunəsi yaradırıq və modelin hiperparametrlərini təyin edirik: order=(1, 0, 0) burada p = 1, d = 0, q = 0 deməkdir və seasonal_order=(0, 1, 1, 7) isə P = 0, D = 1, Q = 1 və s = 7 mənasını verir. Statsmodels API-sinin Scikit-Learn API-sindən bir qədər fərqli olduğunu unutmayın, çünki burada məlumatları fit() metoduna ötürmək əvəzinə, model yaradılan zaman məlumatlar modelə daxil edilir.

- Daha sonra modeli fit() metodu ilə öyrədirik və 1 iyun 2019-cu il üçün (sabah üçün) proqnoz veririk.

Proqnozlaşdırılan sərnişin sayı 427,759 nəfər olmasına baxmayaraq, həmin günə aid real sərnişin sayı 379,044 nəfər olmuşdur. Bu, 12.9% fərqlə kifayət qədər zəif nəticədir. Əslində bu nəticə sadə (naive) proqnozlaşdırma metodundan pisdir. Belə ki, sadə metod 426,932 nəfər proqnozlaşdırır və 12.6% səhv verir.Bunun sadəcə təsadüfi bir hal olub-olmadığını yoxlamaq üçün eyni kodu dövr (loop) içində mart, aprel və may aylarının hər günü üçün tətbiq edə və bu müddət ərzində proqnozların Orta Absolyut Səhvini (MAE) hesablaya bilərik. Bu yanaşma modelin ümumi performansını daha obyektiv qiymətləndirməyə imkan verəcəkdir.

```
origin, start_date, end_date = "2019-01-01", "2019-03-01", "2019-05-31"
time_period = pd.date_range(start_date, end_date)
rail_series = df.loc[origin:end_date]["rail"].asfreq("D")
y_preds = []
for today in time_period.shift(-1):
    model = ARIMA(rail_series[origin:today],  # train on data up to "today"
                  order=(1, 0, 0),
                  seasonal_order=(0, 1, 1, 7))
    model = model.fit()  # note that we retrain the model every day!
```

```
    y_pred = model.forecast()[0]
    y_preds.append(y_pred)

y_preds = pd.Series(y_preds, index=time_period)
mae = (y_preds - rail_series[time_period]).abs().mean()  # returns 32,040.7
```

Bu daha yaxşıdır! MAE (Orta Absolyut Səhv) təxminən 32.041 nəticə verdi, bu da naiv proqnozla əldə etdiyimiz 42.143 MAE dəyərindən əhəmiyyətli dərəcədə aşağıdır. Buna görə də, model mükəmməl olmasa da, orta hesabla naiv proqnozu böyük fərqlə üstələyir. Bu isə modelin proqnozlarının nə dərəcədə yaxşı olduğunu göstərir.

Bu nöqtədə SARİMA model üçün yaxşı hyperparametr-ləri seçəcəyimiz maraq doğura bilər. Bunun üçün bir neçə metod var, amma ən sadə başa düşülən və tətbiq edilə bilən üsul brute-force yanaşmasıdır — yəni grid search üsulu. Qiymətləndirmək istədiyiniz hər model kombinasiyası üçün, əvvəlki kod nümunəsini təkrar işlədə və sadəcə olaraq hiperparametr dəyərlərini dəyişdirə bilərsiniz. Yaxşı p, q, P və Q dəyərləri adətən çox kiçik olur — ümumiyyətlə 0 ilə 2 arasında, bəzən 5 və ya 6-ya qədər artırıla bilər. d və D dəyərləri isə əksər hallarda 0 və ya 1 olur, bəzən də 2 ola bilər. s isə mövsümi modelin əsas periodunu təmsil edir. Bizim nümunədə həftəlik mövsümi təsir güclü olduğu üçün s = 7 seçilir.

Ən aşağı MAE (Orta Absolyut Səhv) dəyərinə malik model ən yaxşı model sayılır. Amma əlbəttə, məqsədinizə daha uyğun başqa bir metrik də seçə bilərsiniz — məsələn, RMSE, MAPE və ya SMAPE. Və budur, hər şey bu qədər sadədir

Maşın Öyrənməsi Modelləri üçün Məlumatların Hazırlanması

Artıq iki əsas istinad nöqtəmiz var: naiv proqnoz və SARIMA modeli. İndi isə əvvəlki bölmələrdə müzakirə etdiyimiz maşın öyrənməsi modellərindən istifadə edərək zaman serisini proqnozlaşdırmağa çalışacağıq. Başlanğıc olaraq sadalə bir xətti modeldən istifadə edəcəyik.Məqsədimiz son 8 həftənin məlumatlarına (yəni 56 günə) əsaslanaraq, sabahkı (t+1) günün sərnişin sayını proqnozlaşdırmaqdır. Buna görə də modelimizə daxil olan girişlər ardıcıllıqlar (sequences) olacaq — istehsalat (production) mərhələsində adətən gündə bir ardıcıllıq istifadə olunacaq.Hər bir ardıcıllıq 56 dəyərdən yəni t – 55-dən t-yə qədər olan zaman addımları dəyərindən ibarət olacaq.

Hər bir giriş ardıcıllığı (sequence) üçün model yalnız bir çıxış dəyəri verəcək:
→ bu da t + 1 zaman addımı üçün proqnozdur.

Bəs təlim məlumatı kimi nə istifadə edəcəyik? Bax əsas məqam budur: keçmişdəki hər bir 56 günlük pəncərəni(window) təlim məlumatı kimi istifadə edəcəyik, və hər pəncərə üçün hədəf isə həmin pəncərədən dərhal sonra gələn dəyər olacaq.

Keras-da tf.keras.utils.timeseries_dataset_from_array() adlı çox faydalı bir köməkçi funksiya var, hansı ki, təlim dəstinin hazırlanmasında bizə kömək edir.

Bu funksiya zaman seriyasını giriş kimi qəbul edir və istədiyimiz uzunluqdakı bütün pəncərələri (windows) və onların uyğun hədəflərini (targets) ehtiva edən tf.data.Dataset (bu, 13-cü fəsildə təqdim olunub) yaradır.

Məsələn, aşağıdakı nümunə zaman seriyası olaraq 0-dan 5-ə qədər olan ədədlər götürür və uzunluğu 3 olan bütün pəncərələri ilə birlikdə uyğun hədəfləri yaradır, sonra isə bunları hər partiyada 2 nümunə olmaqla qruplaşdırır:

```
import tensorflow as tf

my_series = [0, 1, 2, 3, 4, 5]
my_dataset = tf.keras.utils.timeseries_dataset_from_array(
    my_series,
    targets=my_series[3:],  # the targets are 3 steps into the future
    sequence_length=3,
    batch_size=2
)
```

Gəlin bu dataset-in daxilindəkiləri birlikdə yoxlayaq:

```
>>> list(my_dataset)
[(<tf.Tensor: shape=(2, 3), dtype=int32, numpy=
 array([[0, 1, 2],

        [1, 2, 3]], dtype=int32)>,
  <tf.Tensor: shape=(2,), dtype=int32, numpy=array([3, 4], dtype=int32)>),
 (<tf.Tensor: shape=(1, 3), dtype=int32, numpy=array([[2, 3, 4]], dtype=int32)>,
  <tf.Tensor: shape=(1,), dtype=int32, numpy=array([5], dtype=int32)>)]
```

Datasetdəki hər nümunə 3 uzunluğunda bir pəncərədir və onun uyğun hədəfi (yəni, həmin pəncərədən dərhal sonra gələn dəyər) ilə birlikdə təqdim olunur.

Bu pəncərələr və onların hədəfləri belədir:

- Pəncərə: [0, 1, 2] → Hədəf: 3
- Pəncərə: [1, 2, 3] → Hədəf: 4
- Pəncərə: [2, 3, 4] → Hədəf: 5

Ümumilikdə 3 pəncərə olduğu üçün və batch ölçüsü 2 olduğu halda, son batchdə yalnız 1 pəncərə olur (çünki 3, 2-nin tam çoxluğu deyil).

Eyni nəticəni əldə etməyin başqa bir yolu isə tf.data kitabxanasının Dataset sinfinin window() metodundan istifadə etməkdir.Bu üsul bir az daha mürəkkəbdir, amma tam nəzarət imkanı verir ki, bu da bu fəsildə

daha sonra çox faydalı olacaq. Gəlin necə işlədiyinə baxaq.window() metodu öz-özlüyündə pəncərə datasetləri toplusu olan bir dataset qaytarır, yəni:

```
>>> for window_dataset in tf.data.Dataset.range(6).window(4, shift=1):
...     for element in window_dataset:
...         print(f"{element}", end=" ")
...     print()
...
0 1 2 3
1 2 3 4
2 3 4 5
3 4 5
4 5
5
```

Bu nümunədə datasetdə altı pəncərə var və hər biri əvvəlkindən bir addım irəlidə yerləşir (yəni, hər pəncərə bir vahid sürüşdürülüb).Lakin, son üç pəncərə kiçik ölçüdədir, çünki zaman seriyası sona çatıb və tam uzunluqda pəncərə yaratmaq mümkün olmayıb. Ümumiyyətlə, bu kiçik pəncərələrdən xilas olmaq üçün window() metoduna drop_remainder=True arqumentini ötürmək lazımdır.

window() metodu iç-içə (nested) dataset qaytarır, sanki siyahıların siyahısı kimi struktura malikdir.Bu, hər bir pəncərə üzərində ayrıca əməliyyatlar (məsələn, shuffle() və ya batch() kimi dataset metodlarını) çağırmaq istəyəndə faydalıdır. Lakin, bu iç-içə datasetləri birbaşa təlimdə istifadə etmək mümkün deyil, çünki modelimizin giriş olaraq datasetləri yox, tensorlar gözləyir.

Buna görə, flat_map() metodunu çağırmalıyıq: bu metod iç-içə (nested) datasetləri düz (flat) datasetə çevirir — yəni, tensorlardan ibarət datasetə, datasetlərdən deyil. Məsələn, əgər {1, 2, 3} tensorlardan ibarət bir dataset-dirsə və iç-içə dataset { {1, 2}, {3, 4, 5, 6} } şəklindədirsə, flat_map() nəticəsində {1, 2, 3, 4, 5, 6} kimi düz bir dataset əldə edirik.

Əlavə olaraq, flat_map() metodu arqument kimi bir funksiya qəbul edir və bu funksiya, düzləşdirmədən əvvəl iç-içə verilmiş məlumat toplusundakı hər bir dataset-i çevirməyə imkan verir. Məsələn, flat_map() metoduna lambda ds: ds.batch(2) funksiyasını ötürsəniz, onda iç-içə strukturda olan {{1, 2}, {3, 4, 5, 6}} məlumat toplusu düz verilənlər toplusuna –

[{1, 2}, {3, 4}, {5, 6}] -çevriləcəkdir.
Bu, hər biri ölçüsü 2 olan 3 tensordan ibarət bir verilənlər toplusudur.

Bunu nəzərə alaraq, məlumat dəstimizi düzəltməyə hazırıq:

```
>>> dataset = tf.data.Dataset.range(6).window(4, shift=1, drop_remainder=True)
>>> dataset = dataset.flat_map(lambda window_dataset: window_dataset.batch(4))
>>> for window_tensor in dataset:
...     print(f"{window_tensor}")
...
[0 1 2 3]
[1 2 3 4]
[2 3 4 5]
```

Hər bir pəncərə məlumat toplusu dəqiq olaraq dörd elementdən ibarət olduğuna görə, batch(4) funksiyasının pəncərə üzərində çağırılması 4 ölçülü tək bir tensör yaradır. Möhtəşəmdir! Artıq tensor kimi təmsil olunan ardıcıl pəncərələri özündə saxlayan bir məlumat toplusuna sahibik.Məlumat toplusundan pəncərələri çıxarmağı asanlaşdırmaq üçün kiçik bir köməkçi funksiya yaradaq:

```
def to_windows(dataset, length):
    dataset = dataset.window(length, shift=1, drop_remainder=True)
    return dataset.flat_map(lambda window_ds: window_ds.batch(length))
```

Sonuncu addım isə, map() funksiyasından istifadə edərək hər bir pəncərəni giriş və hədəflərə bölmək üsuldur.Biz həmçinin nəticədə yaranan pəncərələri 2 ölçülü dəstlərə qruplaşdıra bilərik:

```
>>> dataset = to_windows(tf.data.Dataset.range(6), 4)  # 3 inputs + 1 target = 4
>>> dataset = dataset.map(lambda window: (window[:-1], window[-1]))
>>> list(dataset.batch(2))
[(<tf.Tensor: shape=(2, 3), dtype=int64, numpy=
  array([[0, 1, 2],

         [1, 2, 3]])>,
  <tf.Tensor: shape=(2,), dtype=int64, numpy=array([3, 4])>),
 (<tf.Tensor: shape=(1, 3), dtype=int64, numpy=array([[2, 3, 4]])>,
  <tf.Tensor: shape=(1,), dtype=int64, numpy=array([5])>)]
```

Gördüyünüz kimi, indi artıq əvvəl timeseries_dataset_from_array() funksiyası ilə əldə etdiyimiz nəticə ilə eyni nəticəni almış olduq (bir az daha çox əziyyət tələb olundu, amma tezliklə buna dəyəcək).

İndi isə təlimə başlamazdan əvvəl məlumatlarımızı üç hissəyə bölməliyik: təlim dövrü, doğrulama (validation) dövrü və test dövrü. Hələlik dəmir yolu sərnişindaşımasına (rail ridership) fokuslanacağıq. Dəyərləri 0–1 aralığına yaxın saxlamaq üçün onları bir milyonla bölərək miqyasını kiçildəcəyik. Bu, həm defolt çəkilərin (weights) başlanğıc dəyərləri, həm də öyrənmə əmsalı (learning rate) ilə daha yaxşı işləyəcək.

```
rail_train = df["rail"]["2016-01":"2018-12"] / 1e6
rail_valid = df["rail"]["2019-01":"2019-05"] / 1e6
rail_test = df["rail"]["2019-06":] / 1e6
```

Qeyd

Zaman seriyası ilə işləyərkən ümumiyyətlə məlumatları zaman üzrə bölmək istəyirsiniz. Ancaq bəzi hallarda digər ölçülər üzrə bölmək mümkün ola bilər, bu da sizə daha uzun bir təlim dövrü deməkdir. Məsələn, əgər 2001–2019 illəri arasındakı 10,000 şirkətin maliyyə sağlamlığı haqqında məlumatınız varsa, bu məlumatı müxtəlif şirkətlər üzrə bölə bilərsiniz. Amma çox güman ki, bu şirkətlərin bir çoxu bir-biri ilə güclü korrelyasiyaya sahibdir (məsələn, bütöv iqtisadi sektorlar birlikdə yuxarı və ya aşağı hərəkət edə bilər) və əgər təlim və test dəstində korrelyasiya olan şirkətlər varsa, test dəstiniz o qədər də fayda verməyəcək, çünki ümumiləşdirmə xətasının ölçüsü optimistik şəkildə azaldılmış (yəni pozitiv meylli) olacaq.

İndi isə timeseries_dataset_from_array() funksiyasından istifadə edərək təlim və doğrulama üçün məlumat dəstləri yaradaq. Qradient enişi (gradient descent) təlim dəstindəki nümunələrin müstəqil və eyni paylanmaya malik olmasını (IID) gözlədiyi üçün, 4-cü fəsildə gördüyümüz kimi, təlim pəncərələrini (windows) qarışdırmaq üçün shuffle=True arqumentini təyin etməliyik (amma onların içindəkiləri qarışdırmamalıyıq).

```
seq_length = 56
train_ds = tf.keras.utils.timeseries_dataset_from_array(
    rail_train.to_numpy(),


    targets=rail_train[seq_length:],
    sequence_length=seq_length,
    batch_size=32,
    shuffle=True,
    seed=42
)
valid_ds = tf.keras.utils.timeseries_dataset_from_array(
    rail_valid.to_numpy(),
    targets=rail_valid[seq_length:],
    sequence_length=seq_length,
    batch_size=32
)
```

İndi isə istədiyimiz reqressiya modelini qurmağa və təlim etməyə hazırıq!

Xətti Modeldən istifadə edərək proqnozlaşdırma

Əvvəlcə sadə xətti model ilə başlayaq.Biz Huber Loss istifadə edəcəyik, adətən MAE-ni (Orta Absolyut Səhv) birbaşa minimallaşdırmaqdan daha yaxşı işləyir, 10-cu fəsildə müzakirə olunduğu kimi. Biz həmçinin erkən dayandırmadan (early stopping) istifadə edəcəyik:

```
tf.random.set_seed(42)
model = tf.keras.Sequential([
    tf.keras.layers.Dense(1, input_shape=[seq_length])
])
early_stopping_cb = tf.keras.callbacks.EarlyStopping(
    monitor="val_mae", patience=50, restore_best_weights=True)
opt = tf.keras.optimizers.SGD(learning_rate=0.02, momentum=0.9)
model.compile(loss=tf.keras.losses.Huber(), optimizer=opt, metrics=["mae"])
history = model.fit(train_ds, validation_data=valid_ds, epochs=500,
                    callbacks=[early_stopping_cb])
```

Bu model təxminən 37,866 doğrulama MAE dəyərinə çatır (sənin nəticən fərqli ola bilər). Bu, sadə proqnozlaşdırmadan daha yaxşıdır, amma SARIMA modelindən zəifdir.
RNN ilə daha yaxşı nəticə əldə edə bilərikmi? Gəlin baxaq!

Sadə RNN-dən İstifadə Edərək Proqnozlaşdırma

Gəlin, içində bir təkrarlanan qat və yalnız bir təkrarlanan neyron olan ən sadə RNN-i sınayaq, 15-1-ci şəklində gördüyümüz kimi:

```
model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(1, input_shape=[None, 1])
])
```

Keras-dakı bütün təkrarlanan (recurrent) qatlar 3D girişlər gözləyir: [batch ölçüsü, zaman addımları, miqyaslılıq] formasında, burada miqyaslılıq (dimensionality) tək dəyişənli zaman seriyası üçün 1, çox dəyişənli zaman seriyası üçün isə daha çox olur. Yadda saxla ki, input_shape arqumenti birinci ölçünü (yəni batch ölçüsünü) nəzərə almır və təkrarlanan qatlar istənilən uzunluqda giriş ardıcıllığını qəbul edə bildiyi üçün ikinci ölçünü None olaraq təyin edə bilərik, yəni "istənilən ölçü". Son olaraq, tək dəyişənli zaman seriyası ilə işlədiyimiz üçün son ölçünün ölçüsü 1 olmalıdır. Buna görə input_shape olaraq [None, 1] göstərdik: bu, "istənilən uzunluqda tək dəyişənli ardıcıllıqlar" deməkdir. Qeyd etmək lazımdır ki, datasetlər əslində [batch ölçüsü, zaman addımları] ölçüsündə girişlər əlavə edir, yəni son ölçü olan 1-i qaçırdırıq, amma Keras bu ölçünü bizim üçün avtomatik əlavə edəcək qədər qayğıkeşdir.

Bu model əvvəlcə gördüyümüz kimi işləyir: başlanğıc vəziyyət $\mathbf{h}_{(init)}$ sıfıra (0) bərabər olaraq təyin olunur və bu, birinci zaman addımındakı giriş dəyəri $\mathbf{x}_{(0)}$ ilə birlikdə təkrarlanan neyrona ötürülür. Neyron bu dəyərlərin çəkili cəmini və bias terminini hesablayır, sonra isə nəticəyə aktivasiya funksiyası tətbiq edir — standart olaraq bu, hiperbolik tangens (tanh) funksiyasıdır. Alınan nəticə birinci çıxış $\mathbf{y}_{(0)}$ olur. Sadə RNN-də bu çıxış həm də yeni vəziyyət $\mathbf{h}_{(0)}$ kimi təyin olunur.

Ən sonda qat yalnız sonuncu dəyəri çıxış kimi verir: bizim halda ardıcıllıqlar 56 addımdan ibarətdir, ona görə son dəyər $\mathbf{y}_{(55)}$ olur. Bütün bu əməliyyatlar batch-dəki hər bir ardıcıllıq üçün eyni anda həyata keçirilir; burada batch ölçüsü 32-dir.

> **Qeyd**
>
> Standart olaraq, Keras-da rekurrent qatlar yalnız son çıxışı qaytarır. Onların hər zaman addımı üçün bir çıxış qaytarmasını istəyirsinizsə, return_sequences=True parametrini təyin etməlisiniz — bunu bir azdan görəcəksiniz.

Beləliklə, bu bizim ilk rekurrent modelimiz oldu! Bu, sequence-to-vector (ardıcıllıqdan vektora) modeldir. Çünki yalnız bir çıxış neyronu var, nəticədə çıxış vektorunun ölçüsü 1 olur.

İndi bu modeli əvvəlki model kimi compile, train və evaluate etsəniz, görəcəksiniz ki, bu model heç yaxşı işləmir: onun doğrulama MAE (Mean Absolute Error) göstəricisi 100,000-dən çoxdur! Bu isə gözlənilən idi, iki əsas səbəbə görə:

1. Model cəmi bir rekurrent neyrona malikdir, yəni hər zaman addımında proqnoz vermək üçün istifadə edə biləcəyi tək məlumat – həmin anın giriş dəyəri və əvvəlki zaman addımındakı çıxış dəyəridir. Bu isə çox məhduddur! Başqa sözlə, bu RNN-in yaddaşı sadəcə bir ədəddən – yəni əvvəlki çıxışdan ibarət olduğu üçün olduqca zəifdir. İndi isə gəlin modelin neçə parametri olduğunu sayaq: bir rekurrent neyron və iki giriş dəyəri olduğundan, modeldə cəmi üç parametr

var (iki weight və bir bias). Bu isə bu zaman sırası üçün yetərli deyil. Əksinə, əvvəlki model eyni anda 56 əvvəlki dəyəri görə bilirdi və ümumilikdə isə 57 parametri var idi.

2. Zaman sırasındakı dəyərlər 0 ilə təxminən 1.4 arasında dəyişir, lakin modeldəki rekurrent qatın defolt aktivasiya funksiyası tanh olduğundan, çıxış yalnız –1 ilə +1 arasında ola bilər. Bu halda isə 1.0–1.4 aralığındakı dəyərləri proqnozlaşdırmaq qeyri-mümkündür.

Gəlin bu iki problemi aradan qaldıraq: daha böyük bir rekurrent qat ilə model quracağıq — bu qat 32 rekurrent neyronu əhatə edəcək. Üstəlik, onun üzərinə aktivasiya funksiyası olmayan, tək çıxış neyronlu bir dense (tam əlaqəli) çıxış qatı əlavə edəcəyik.

- Rekurrent qat hər bir zaman addımından digərinə daha çox məlumat daşıya biləcək.
- Dense çıxış qatı isə heç bir dəyər aralığı məhdudiyyəti olmadan ,32 ölçülü son çıxışı 1 ölçüyə proyeksiya edəcək.

```
univar_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, input_shape=[None, 1]),
    tf.keras.layers.Dense(1)  # no activation function by default
])
```

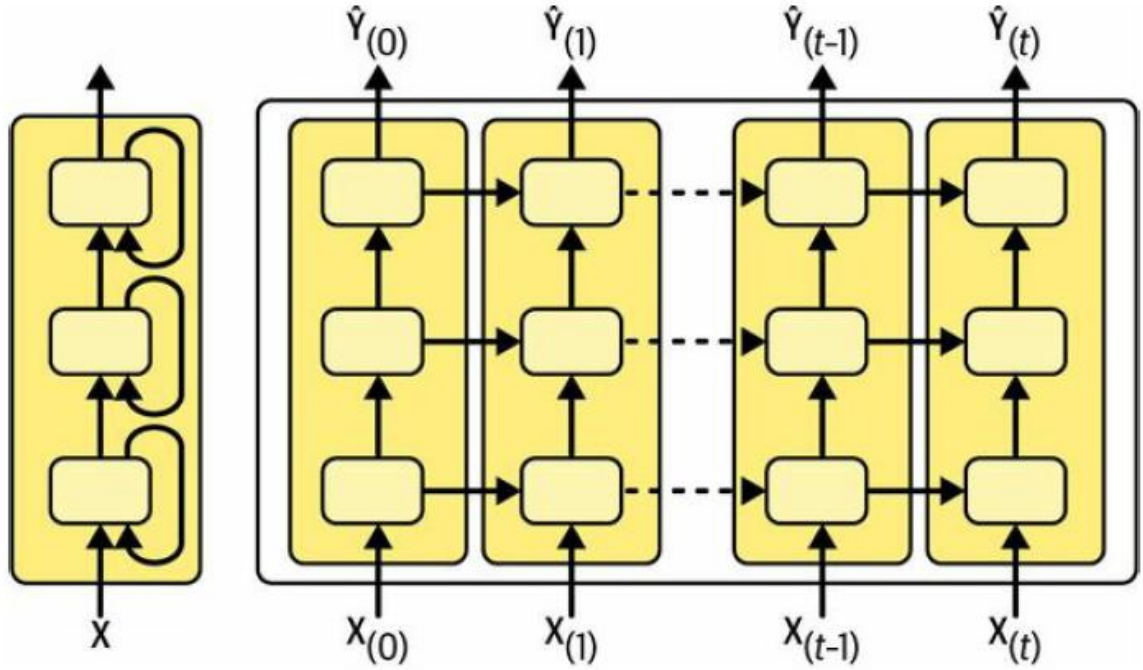
İndi bu modeli əvvəlki kimi compile (tərtib edək), fit (öyrət) və evaluate (qiymətləndir) etsən, görəcəksən ki, onun validasiya MAE göstəricisi 27,703-ə çatır. Bu, indiyədək öyrətdiyimiz modellər arasında ən yaxşısıdır və hətta SARIMA modelindən də yaxşı nəticə göstərir — yəni olduqca yaxşı irəliləmişik!

Tövsiyə

Biz yalnız zaman seriyasını normallaşdırmışıq, trendləri və mövsümləri çıxarmamışıq, amma model hələ də yaxşı nəticə göstərir. Bu əlverişlidir, çünki perspektivli modelləri tez bir zamanda yoxlamaq imkanı verir və preprocessing barədə çox narahat olmağa ehtiyac qalmır. Lakin, ən yaxşı performansı əldə etmək üçün məsələn, fərqləndirmə (differencing) istifadə edərək zaman seriyasını daha stasionar etməyi arzulaya bilərsiniz.

Dərin RNN istifadə edərək proqnozlaşdırma

Çox vaxt bir neçə hüceyrə qatını üst-üstə yığmaq (stack) adi haldır, necə ki, Təsvir 15-10-da göstərilmişdir.Bu, sizə dərin RNN (Deep RNN) əldə etməyə imkan verir.



Təsvir 15-10. Dərin RNN (solda), zaman üzrə unrolled (açıq) olunmuş formada (sağda)

Keras ilə dərin RNN-in implementasiyası sadədir: sadəcə rekurrent qatları (recurrent layers) üst-üstə yığın. Aşağıdakı nümunədə biz üç SimpleRNN qatı istifadə edirik (amma əvəzinə istənilən digər rekurrent qat – məsələn, LSTM və ya GRU də istifadə oluna bilər; bunları az sonra müzakirə edəcəyik).

Birinci iki qat sequence-to-sequence (girişdən çıxışa ardıcıllıq verən) qatdır, sonuncu isə sequence-to-vector (ardıcıllıqdan vektora) qatdır. Sonda isə Dense qatı modelin proqnozunu verir(bunu vector-to-vector qat kimi də təsəvvür edə bilərsiniz).

Beləliklə, bu model Təsvir 15-10-da göstərilən modelə bənzəyir, sadəcə olaraq aradakı çıxışlar $\hat{\mathbf{Y}}_{(0)} \rightarrow \hat{\mathbf{Y}}_{(t-1)}$ nəzərə alınmır və ən yuxarıda Dense qatı $\hat{\mathbf{Y}}_{(t)}$ yerləşdirilir. Bu qat isə faktiki proqnozu çıxış kimi verir.

```
deep_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, return_sequences=True, input_shape=[None, 1]),
    tf.keras.layers.SimpleRNN(32, return_sequences=True),
    tf.keras.layers.SimpleRNN(32),
    tf.keras.layers.Dense(1)
```

XƏBƏRDARLIQ

Bütün rekurrent qatlarda (sonuncudan başqa, əgər sən yalnız son çıxışla maraqlanırsansa) return_sequences=True olaraq təyin etməyi unutma.

Əgər bu parametri hansısa rekurrent qat üçün qoymağı unutursansa:

- həmin qat yalnız son zaman addımının çıxışını (2D massiv) qaytaracaq,
- halbuki növbəti qat bütün zaman addımlarının çıxışlarını (3D massiv) gözləyir. Nəticədə, növbəti rekurrent qat **"**sən ona ardıcıllıq (sequence) vermirsən**"** deyə şikayət edəcək.

Əgər bu modeli məşq etdirib qiymətləndirsən, görəcəksən ki, o, təxminən 31,211 MAE-ə çatır. Bu, hər iki baza göstəricisindən yaxşıdır, amma bizim "daha dayaz" (az qatlı) RNN-i üstələyə bilmir. Görünür, bu RNN bizim tapşırıq üçün bir az çox böyükdür.

Çoxdəyişənli (Multivariate) Zaman Sıralarının Proqnozlaşdırılması

Süni sinir şəbəkələrinin böyük üstünlüklərindən biri onların elastikliyidir: xüsusilə də, onlar çoxdəyişənli zaman sıraları ilə demək olar ki, memarlığında heç bir dəyişiklik etmədən işləyə bilirlər. Məsələn, gəlin dəmir yolu zaman sırasını həm avtobus , həm də dəmir yolu məlumatlarını giriş kimi istifadə edərək proqnozlaşdıraq. Əslində, gəlin günün tipini (day type) də əlavə edək! Çünki biz həmişə qabaqcadan bilə bilərik ki, sabah iş günüdür, həftəsonudur, yoxsa bayramdır. Bu səbəbdən, günün tipi sırasını bir gün irəli sürüşdürə bilərik, beləliklə modelə sabahın gün tipi giriş kimi verilmiş olacaq. Sadələşdirmək üçün isə biz bu işi Pandas kitabxanası ilə edəcəyik.

```
df_mulvar = df[["bus", "rail"]] / 1e6  # use both bus & rail series as input
df_mulvar["next_day_type"] = df["day_type"].shift(-1)  # we know tomorrow's type
df_mulvar = pd.get_dummies(df_mulvar)  # one-hot encode the day type
```

İndi df_mulvar beş sütundan ibarət DataFrame-dir: avtobus və qatar məlumatları, üstəgəl növbəti günün tipinin one-hot kodlaşdırmasını əks etdirən üç sütun (xatırladaq ki, üç mümkün gün tipi var: W, A və U). Sonra biz əvvəlki kimi davam edə bilərik. Əvvəlcə verilənləri üç dövrə bölürük: təlim, doğrulama və sınaq üçün.

```
mulvar_train = df_mulvar["2016-01":"2018-12"]
mulvar_valid = df_mulvar["2019-01":"2019-05"]
mulvar_test = df_mulvar["2019-06":]
```

Sonra dataset-lər yaradırıq:

```
train_mulvar_ds = tf.keras.utils.timeseries_dataset_from_array(
    mulvar_train.to_numpy(),  # use all 5 columns as input
    targets=mulvar_train["rail"][seq_length:],  # forecast only the rail series
    [...]  # the other 4 arguments are the same as earlier
)
valid_mulvar_ds = tf.keras.utils.timeseries_dataset_from_array(
    mulvar_valid.to_numpy(),
    targets=mulvar_valid["rail"][seq_length:],
    [...]  # the other 2 arguments are the same as earlier
)
```

Və sonda RNN yaradırıq:

```
mulvar_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, input_shape=[None, 5]),
    tf.keras.layers.Dense(1)
])
```

Qeyd edim ki, daha əvvəl qurduğumuz univar_model RNN-dən yeganə fərq giriş formasındadır: hər zaman addımında model bir input alırdısa indi isə beş giriş alır. Bu model əslində 22,062-lik doğrulama MAE nəticəsinə çatır. Artıq nəzərəçarpacaq dərəcədə irəliləyirik!

Əslində RNN-in həm avtobus, həm də qatar sərnişin axınını proqnozlaşdırması da o qədər çətin deyil. Bunun üçün sadəcə datasetləri yaradarkən hədəfləri dəyişmək lazımdır:

təlim dəsti üçün mulvar_train[["bus","rail"]][seq_length:],

doğrulama dəsti üçün isə mulvar_valid[["bus", "rail"]][seq_length:]

təyin edilməlidir.

Həmçinin, çıxışdakı Dense qatına bir neyron da əlavə olunmalıdır, çünki model artıq iki proqnoz verməlidir: birincisi sabahkı avtobus sərnişin axını, digəri isə qatar üçün. Bütün məsələ budur!

10-cu fəsildə müzakirə etdiyimiz kimi, bir neçə əlaqəli tapşırıq üçün vahid modeldən istifadə çox vaxt hər tapşırıq üçün ayrıca model qurmaqdan daha yaxşı nəticə verir. Çünki bir tapşırıq üçün öyrənilən xüsusiyyətlər digərlərinə də faydalı ola bilər, həm də birdən çox tapşırıqda yaxşı işləmək məcburiyyəti modelin overfitting etməsinin qarşısını alır (bu bir növ reguliyaziyadır). Ancaq bu, tapşırıqdan asılıdır və məhz bu halda, həm avtobus, həm də qatar sərnişin axınını eyni anda proqnozlaşdıran multitapşırıq RNN, hər biri üçün ayrıca qurulmuş modellər qədər yaxşı işləmir (bütün beş sütunu giriş kimi istifadə etsək belə). Yenə də, bu model qatar üçün 25,330, avtobus üçün isə 26,369-luq doğrulama MAE nəticəsi əldə edir ki, bu da kifayət qədər yaxşıdır.

Bir neçə zaman addımını qabaqcadan proqnozlaşdırmaq

İndiyə qədər biz yalnız növbəti zaman addımındakı dəyəri proqnozlaşdırmışıq, amma hədəfləri uyğun şəkildə dəyişməklə bir neçə addım sonranı da asanlıqla proqnozlaşdıra bilərik (məsələn, sərnişin sayını 2 həftə sonra proqnozlaşdırmaq üçün hədəfi 1 gün əvəzinə 14 gün sonranın dəyəri olaraq dəyişdirə bilərik).

Amma əgər biz növbəti 14 dəyəri proqnozlaşdırmaq istəyiriksə, seçimlərdən biri belədir: Əvvəl öyrətdiyimiz univar_model RNN-i götürək, qatar zaman seriyası üçün növbəti dəyəri proqnozlaşdıraq, sonra həmin proqnozlaşdırılan dəyəri girişlərə əlavə edək, elə bil ki, bu dəyər həqiqətən baş verib. Sonra modeli yenidən istifadə edərək növbəti dəyəri proqnozlaşdırırıq və belə davam edirik.

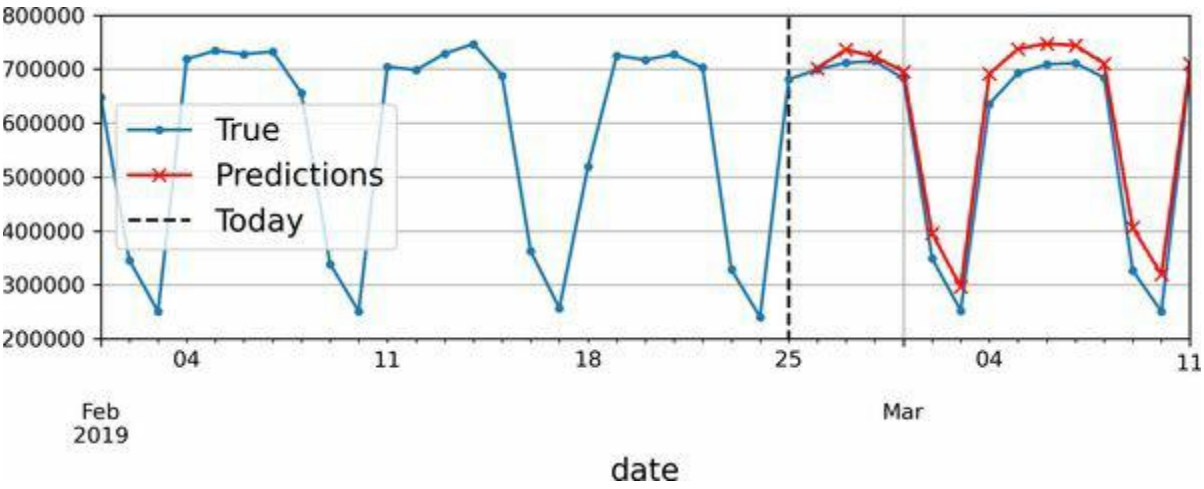
Bu üsul aşağıdakı kod nümunəsində göstərilir:

```
import numpy as np

X = rail_valid.to_numpy()[np.newaxis, :seq_length, np.newaxis]
for step_ahead in range(14):
    y_pred_one = univar_model.predict(X)
    X = np.concatenate([X, y_pred_one.reshape(1, 1, 1)], axis=1)
```

Bu kodda biz doğrulama dövrünün ilk 56 günü üçün qatar sərnişin sayını götürürük və məlumatı [1, 56, 1] ölçülü NumPy array-ə çeviririk (xatırladaq ki, təkrarlayan qatlar 3D giriş gözləyir). Sonra modeli təkrar-təkrar istifadə edərək növbəti dəyəri proqnozlaşdırırıq və hər proqnozu giriş seriyasına vaxt oxu üzrə (axis=1) əlavə edirik. Nəticədə əldə olunan proqnozlar Təsvir 15-11-də təsvir olunub.

XƏBƏRDARLIQ

Əgər model bir zaman addımında səhv edirsə, növbəti zaman addımlarının proqnozları da təsirlənir: səhvlər yığılmağa meyllidir. Buna görə, bu texnikanı yalnız az sayda addım üçün istifadə etmək daha məqsədəuyğundur.



Təsvir 15-11. 14 addımı hər dəfə 1 addım olaraq qabaqcadan proqnozlaşdırmaq

İkinci seçim isə RNN-i birbaşa növbəti 14 dəyəri bir anda proqnozlaşdırmaq üçün öyrətməkdir. Hələ də sequence-to-vector modelindən istifadə edə bilərik, amma çıxış 1 dəyər əvəzinə 14 dəyər olacaq. Bunun üçün əvvəlcə hədəfləri növbəti 14 dəyəri ehtiva edən vektorlar kimi dəyişdirməliyik.

Bunu etmək üçün timeseries_dataset_from_array() funksiyasından yenidən istifadə edə bilərik, amma bu dəfə hədəflər olmadan (targets=None) və daha uzun ardıcıllıqlarla (seq_length + 14) dataset yaratmasını istəyirik. Sonra map() metodundan istifadə edərək, hər batch ardıcıllığa xüsusi bir funksiya tətbiq edə bilərik, onları giriş və hədəflərə ayırmaq üçün.

Bu nümunədə, çoxölçülü zaman seriyasını giriş kimi istifadə edirik (bütün beş sütunu), və növbəti 14 gün üçün (qatar) sərnişin sayını proqnozlaşdırırıq.

```
def split_inputs_and_targets(mulvar_series, ahead=14, target_col=1):
    return mulvar_series[:, :-ahead], mulvar_series[:, -ahead:, target_col]

ahead_train_ds = tf.keras.utils.timeseries_dataset_from_array(
    mulvar_train.to_numpy(),
    targets=None,
    sequence_length=seq_length + 14,
    [...]  # the other 3 arguments are the same as earlier
).map(split_inputs_and_targets)
ahead_valid_ds = tf.keras.utils.timeseries_dataset_from_array(
    mulvar_valid.to_numpy(),
    targets=None,
    sequence_length=seq_length + 14,
    batch_size=32
).map(split_inputs_and_targets)
```

İndi çıxış qatının 1 əvəzinə 14 neyron-a malik olmasına ehtiyac var:

```
ahead_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, input_shape=[None, 5]),
    tf.keras.layers.Dense(14)
])
```

Modeli train etdikdən sonra növbəti **14 dəyəri bir anda** proqnozlaşdırmaq olar bunun kimi:

```
X = mulvar_valid.to_numpy()[np.newaxis, :seq_length]  # shape [1, 56, 5]
Y_pred = ahead_model.predict(X)  # shape [1, 14]
```

Bu yanaşma kifayət qədər yaxşı işləyir. Növbəti gün üçün proqnozları əlbəttə ki, 14 gün sonranın proqnozlarından daha dəqiqdir, amma əvvəlki yanaşma kimi error-ları yığmır. Bununla belə, sequence-to-sequence (seq2seq) modelindən istifadə etməklə daha yaxşı nəticə əldə etmək olar.

Sequence-to-Sequence Model ilə Proqnozlaşdırma

Modeli son zaman addımında növbəti 14 dəyəri proqnozlaşdırmağı öyrətmək əvəzinə, onu hər bir zaman addımında növbəti 14 dəyəri proqnozlaşdırmaq üçün öyrədə bilərik. Başqa sözlə, bu sequence-to-vector RNN-i sequence-to-sequence RNN-ə çevirə bilərik.

Bu texnikanın üstünlüyü odur ki, loss funksiyası yalnız son zaman addımındakı çıxış üçün deyil, hər bir zaman addımındakı çıxış üçün də bir termini özündə saxlayacaq.

Bu o deməkdir ki, modeldən keçən səhv gradientləri çoxalacaq və onlar zaman üzrə çox uzun məsafə boyunca axmaq məcburiyyətində qalmayacaq, çünki yalnız sonuncudan yox, hər zaman addımının çıxışından gələcəklər. Bu həm təlimi stabilləşdirəcək, həm də sürətləndirəcək.

Aydın olması üçün belə deyək: zaman addımı 0-da, model 1-dən 14-ə qədər olan zaman addımlarının proqnozlarını ehtiva edən bir vektor çıxaracaq, sonra zaman addımı 1-də, model 2-dən 15-ə qədər olan proqnozları verəcək və s.

Başqa sözlə, hədəflər ardıcıl pəncərələrdən ibarət ardıcıllıqlardır, hər zaman addımında bir addım sağa sürüşdürülür. Hədəf artıq sadə bir vektor deyil, girişlərlə eyni uzunluqda bir ardıcıllıqdir və hər addımda 14 ölçülü vektor ehtiva edir.

Dataseti hazırlamaq sadə iş deyil, çünki hər nümunənin girişi bir pəncərə, çıxışı isə pəncərələr ardıcıllığıdır.Bunun yolu, əvvəl yaratdığımız to_windows() funksiyasını iki dəfə ardıcıl istifadə etməkdir, beləliklə ardıcıl pəncərələrdən ibarət pəncərələr əldə edirik.Məsələn, 0-dan 6-ya qədər ədədlərdən ibarət seriyanı uzunluğu 3 olan 4 ardıcıl pəncərədən ibarət dataset-ə çevirmək olar:

```
>>> my_series = tf.data.Dataset.range(7)
>>> dataset = to_windows(to_windows(my_series, 3), 4)
>>> list(dataset)
[<tf.Tensor: shape=(4, 3), dtype=int64, numpy=
 array([[0, 1, 2],
        [1, 2, 3],
        [2, 3, 4],
        [3, 4, 5]])>,
 <tf.Tensor: shape=(4, 3), dtype=int64, numpy=
 array([[1, 2, 3],


 [2, 3, 4],
 [3, 4, 5],
 [4, 5, 6]])>]
```

İndi biz map() metodundan istifadə edərək bu pəncərələr ardıcıllığını (windows of windows) girişlər və hədəflər olaraq bölə bilərik.

```
>>> dataset = dataset.map(lambda S: (S[:, 0], S[:, 1:]))
>>> list(dataset)
[(<tf.Tensor: shape=(4,), dtype=int64, numpy=array([0, 1, 2, 3])>,
  <tf.Tensor: shape=(4, 2), dtype=int64, numpy=
  array([[1, 2],
         [2, 3],
         [3, 4],
         [4, 5]])>),
 (<tf.Tensor: shape=(4,), dtype=int64, numpy=array([1, 2, 3, 4])>,
  <tf.Tensor: shape=(4, 2), dtype=int64, numpy=
  array([[2, 3],
         [3, 4],
         [4, 5],
         [5, 6]])>)]
```

İndi datasetin girişləri uzunluğu 4 olan ardıcıllıqlardan, hədəflər isə hər zaman addımı üçün növbəti iki addımı ehtiva edən ardıcıllıqlardan ibarətdir.

Məsələn, ilk giriş ardıcıllığı [0, 1, 2, 3]-dür və onun müvafiq hədəfləri yəni hər zaman addımı üçün növbəti iki dəyər [[1, 2], [2, 3], [3, 4], [4, 5]]-dir.

Əgər mənim kimi düşünürsünüzsə, bunu anlamaq üçün bir neçə dəqiqə lazım ola bilər. Sakitcə və addım-addım anlayın!

QEYD

Təəccüblü görünə bilər ki, hədəflərdə girişlərdə artıq olan dəyərlər var. Bu "aldatma" deyil.

Səbəb: hər zaman addımında RNN yalnız keçmiş zaman addımlarını bilir; gələcəyə baxa bilmir. Bu səbəbdən belə modellərə causal (sebəb-nəticə əlaqəli) model deyilir.

Gəlin sequence-to-sequence modeli üçün datasetləri hazırlayan kiçik bir utility funksiyası yaradaq.

Bu funksiya həmçinin shuffle (istəyə bağlı) və batch-ləri də idarə edəcək.

```
def to_seq2seq_dataset(series, seq_length=56, ahead=14, target_col=1,
                       batch_size=32, shuffle=False, seed=None):
    ds = to_windows(tf.data.Dataset.from_tensor_slices(series), ahead + 1)
    ds = to_windows(ds, seq_length).map(lambda S: (S[:, 0], S[:, 1:, 1]))
    if shuffle:
        ds = ds.shuffle(8 * batch_size, seed=seed)
    return ds.batch(batch_size)
```

İndi bu funksiyanı dataset yaratmaq üçün istifadə edəcəyik:

```
seq2seq_train = to_seq2seq_dataset(mulvar_train, shuffle=True, seed=42)
seq2seq_valid = to_seq2seq_dataset(mulvar_valid)
```

Və nəhayət, sequence-to-sequence modelini qura bilərik:

```
seq2seq_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, return_sequences=True, input_shape=[None, 5]),
    tf.keras.layers.Dense(14)
])
```

Bu model əvvəlki modelimizlə demək olar ki, eynidir; yeganə fərq SimpleRNN qatında return_sequences=True qoymağımızdır. Beləliklə, model son zaman addımında tək vektor çıxarmaq əvəzinə, hər addım üçün 32 ölçülü vektor ardıcıllığı çıxaracaq.

Dense qatı ardıcıllıqları giriş kimi idarə etmək üçün kifayət qədər ağıllıdır: hər zaman addımında tətbiq olunur, 32 ölçülü vektoru götürür və 14 ölçülü vektor çıxarır.

Əslində, eyni nəticəni əldə etməyin başqa bir yolu da Conv1D qatından kernel size=1 ilə istifadə etməkdir: Conv1D(14, kernel_size=1).

Tövsiyə

Keras TimeDistributed qatını təklif edir, bu qat hər bir zaman addımında giriş ardıcıllığındakı hər bir vektora istənilən vector-to-vector qatını tətbiq etməyə imkan verir.Bunu effektiv edir: girişləri hər zaman addımını ayrı nümunə kimi qəbul edəcək şəkildə dəyişdirir, sonra çıxışları zaman ölçüsünü bərpa edəcək şəkildə yenidən formatlayır.Bizim halda isə buna ehtiyac yoxdur, çünki Dense qatı artıq ardıcıllıqları giriş kimi dəstəkləyir.

Təlim kodu əvvəlki kimi qalır. Təlim zamanı modelin bütün çıxışları istifadə olunur, amma təlim bitdikdən sonra yalnız son zaman addımının çıxışı vacibdir, qalanlar isə nəzərə alınmır.

Məsələn, beləliklə növbəti 14 gün üçün dəmir yolu sərnişin sayını proqnozlaşdırmaq olar:

```
X = mulvar_valid.to_numpy()[np.newaxis, :seq_length]
y_pred_14 = seq2seq_model.predict(X)[0, -1]  # only the last time step's output
```

Əgər bu modelin t + 1 üçün proqnozlarını qiymətləndirsəniz, validation MAE 25,519 olacaq. t + 2 üçün isə 26,274, və model gələcəyə daha uzağa proqnoz verdikcə performans tədricən azalır.

t + 14 üçün MAE 34,322-d

Tövsiyə

Hər iki yanaşmanı birləşdirərək bir neçə addım qabağı proqnozlaşdıra bilərsiniz:Məsələn, 14 gün qabağa proqnoz verən bir model öyrədin, sonra onun çıxışını girişlərə əlavə edin və modeli yenidən işlədərək növbəti 14 gün üçün proqnozlar alın. Bu prosesi təkrar etmək də mümkündür.

Uzun Ardıcıllıqlarla İşləmək

Uzun ardıcıllıqlarda RNN-i öyrətmək üçün onu çox sayda zaman addımı boyunca işlətməliyik, bu da unrolled RNN-i çox dərin bir şəbəkəyə çevirir.Hər hansı dərin neyron şəbəkədə olduğu kimi, bu unstable gradients (sabit olmayan gradient) problemi yarada bilər:Fəsil 11 təlim çox uzun çəkə bilər və ya qeyri-sabit ola bilər.Üstəlik, RNN uzun ardıcıllıqları emal edərkən ilk girişləri tədricən unutmağa başlayır.İndi bu iki problemə baxaq, əvvəlcə unstable gradients problemindən başlayaq.

Qeyri-sabit qradientlər probleminə qarşı mübarizə

Dərin neyron şəbəkələrdə qeyri-sabit qradientlər problemini yüngülləşdirmək üçün istifadə etdiyimiz bir çox üsullar RNN-lərdə də tətbiq oluna bilər: məsələn, good parameter initialization, sürətli optimizatorlar, dropout və s. Lakin doğrusal olmayan, saturasiya etməyən aktivasiya funksiyaları (məsələn, ReLU) burada o qədər də faydalı olmaya bilər. Əslində, onlar RNN-in təlim zamanı daha da qeyri-sabit olmasına səbəb ola bilərlər.

Niyə? Tutaq ki, gradient descent çəkiləri elə yeniləyir ki, birinci zaman addımında çıxışları bir qədər artırır. Çünki eyni çəkilər hər zaman addımında istifadə olunur, ikinci zaman addımında çıxışlar da bir qədər arta bilər, üçüncü addımda da və s., nəticədə çıxışlar partlaya bilər — və saturasiya etməyən aktivasiya funksiyası buna mane olmur.

Bu riskləri learning rate istifadə etməklə azalda bilərsiniz, yaxud saturasiya edən aktivasiya funksiyası, məsələn, hyperbolic tangent (tanh) istifadə edə bilərsiniz (buna görə də tanh defolt olaraq seçilir).

Oxşar şəkildə, qradientlərin özləri də partlaya bilər (exploding gradients). Əgər təlim prosesi qeyri-sabitdirsə, qradientlərin böyüklüyünü izləmək istəyə bilərsiniz (məsələn, TensorBoard vasitəsilə) və lazım gələrsə qradient clipping tətbiq etmək olar.

Üstəlik, batch normalization RNN-lərdə dərin feedforward şəbəkələrdə olduğu qədər effektiv istifadə oluna bilməz. Əslində, onu zaman addımları arasında istifadə etmək mümkün deyil, yalnız rekurrent qatlar arasında istifadə oluna bilər.

Daha dəqiq desək, texniki baxımdan bir BN (Batch Normalization) qatını yaddaş hüceyrəsinə hər zaman addımında tətbiq olunması (tezliklə görəcəyiniz kimi) üçün əlavə etmək mümkündür(həm həmin zaman addımı üçün girişlərə, həm də əvvəlki addımdan gələn gizli vəziyyətə). Lakin, eyni BN qatı hər zaman addımında eyni parametrlərlə girişlərin və gizli vəziyyətin faktiki miqyası və offset-i nəzərə alınmadan istifadə olunur. Təcrübədə isə bu yaxşı nəticələr vermir; bunu César Laurent və başqaları 2015-ci ildəki məqalələrində göstərmişdir: müəlliflər aşkar etmişdilər ki, BN yalnız qatın girişlərinə tətbiq olunduqda bir qədər faydalıdır, gizli vəziyyətlərə tətbiq edildikdə isə yox. Başqa sözlə, rekurrent qatlar arasında tətbiq edildikdə (yəni Təsvir 15-10-da vertikal olaraq) bir qədər faydalı olsa da, rekurrent qatların içində (yəni horizontal olaraq) effektiv deyil. Keras-da BN-i qatlar arasında sadəcə hər rekurrent qatın əvvəlinə BatchNormalization qatını əlavə etməklə tətbiq edə bilərsiniz, lakin bu təlimi yavaşıdır və çox kömək etməyə bilər.

RNN-lərdə tez-tez daha yaxşı işləyən başqa bir normallaşdırma forması var: layer normalization. Bu ideya Jimmy Lei Ba və başqaları tərəfindən 2016-cı ildə təqdim olunmuşdur. Layer normalization batch normalization-a çox bənzəyir, lakin batch ölçüsü üzrə normalizasiya əvəzinə, layer normalization xüsusiyyətlər (features) ölçüsü üzrə normalizasiya edir. Üstünlüklərindən biri odur ki, tələb olunan statistikaları hər zaman addımında, hər nümunə üçün müstəqil şəkildə hesablaya bilər. Bu həmçinin o deməkdir ki, layer normalization təlim və test zamanı eyni davranır (BN-dən fərqli olaraq) və BN kimi bütün təlim nümunələri üzrə xüsusiyyət statistikalarını qiymətləndirmək üçün exponential moving average-dən istifadə etməyə ehtiyac yoxdur. BN kimi, layer normalization də hər giriş üçün bir scale və offset parametrlərini öyrənir. RNN-lərdə isə adətən girişlər və gizli vəziyyətlərin xətti kombinasiyasından sonra tətbiq olunur.

Gəlin Keras-dan istifadə edərək sadə bir yaddaş hüceyrəsində layer normalization tətbiq edək. Bunu etmək üçün xüsusi bir yaddaş hüceyrəsi (custom memory cell) təyin etməliyik. Bu, adi bir qat kimidir, lakin onun call() metodu iki arqument qəbul edir: cari zaman addımındakı girişlər və əvvəlki zaman addımından gələn gizli vəziyyətlər.

Qeyd edək ki, states arqumenti bir və ya bir neçə tensoru özündə saxlayan siyahıdır. Sadə RNN hüceyrəsi üçün bu siyahıda yalnız bir tensor olur, o da əvvəlki zaman addımının çıxışına bərabərdir. Lakin digər

hüceyrələrdə bir neçə state tensordan ibarət ola bilər (məsələn, LSTMCell-də həm uzunmüddətli, həm də qısamüddətli state olur, bunu tezliklə görəcəksiniz).

Hüceyrədə həmçinin state_size və output_size atributları olmalıdır. Sadə RNN-də hər ikisi sadəcə units sayına bərabərdir. Aşağıdakı kod xüsusi yaddaş hüceyrəsini implement edir, bu hüceyrə SimpleRNNCell kimi davranacaq, lakin hər zaman addımında layer normalization tətbiq edəcək:

```
class LNSimpleRNNCell(tf.keras.layers.Layer):
    def __init__(self, units, activation="tanh", **kwargs):
        super().__init__(**kwargs)
        self.state_size = units
        self.output_size = units
        self.simple_rnn_cell = tf.keras.layers.SimpleRNNCell(units,
                                                             activation=None)


        self.layer_norm = tf.keras.layers.LayerNormalization()
        self.activation = tf.keras.activations.get(activation)

    def call(self, inputs, states):
        outputs, new_states = self.simple_rnn_cell(inputs, states)
        norm_outputs = self.activation(self.layer_norm(outputs))
        return norm_outputs, [norm_outputs]
```

Gəlin bu kodu addım-addım nəzərdən keçirək:

Bizim LNSimpleRNNCell sinfi tf.keras.layers.Layer sinfindən irs alır, adi bir custom layer kimi.

Constructor (yəni __init__) aşağıdakıları edir:

- Units sayını və istifadə etmək istədiyimiz aktivasiya funksiyasını qəbul edir.
- state_size və output_size atributlarını təyin edir.
- Aktivasiya funksiyasız bir SimpleRNNCell yaradır (çünki linear əməliyyatdan sonra, amma aktivasiya funksiyasından əvvəl layer normalization tətbiq etmək istəyirik).
- Sonra LayerNormalization qatını yaradır və istədiyimiz aktivasiya funksiyasını əldə edir.

call() metodu belə işləyir:

1. Əvvəlcə simpleRNNCell tətbiq olunur; bu, cari girişlər və əvvəlki gizli vəziyyətlərin xətti kombinasiyasını hesablayır və nəticəni iki dəfə qaytarır (SimpleRNNCell-də çıxışlar gizli

vəziyyətlərə bərabərdir, yəni new_states[0] outputs-a bərabərdir və call() metodunun qalan hissəsində new_statesi rahatlıqla nəzərə almaya bilərik).
2. Daha sonra call() metodu layer normalization tətbiq edir, ardınca aktivasiya funksiyasını.
3. Nəticədə outputs iki dəfə qaytarılır: bir dəfə çıxış olaraq, bir dəfə yeni gizli vəziyyət olaraq.

Bu custom hüceyrəni istifadə etmək üçün sadəcə tf.keras.layers.RNN qatını yaratmaq və ona hüceyrə instansiyasını vermək kifayətdir.

```
custom_ln_model = tf.keras.Sequential([
   tf.keras.layers.RNN(LNSimpleRNNCell(32), return_sequences=True,
                       input_shape=[None, 5]),
   tf.keras.layers.Dense(14)
])
```

Oxşar şəkildə, hər zaman addımı arasında dropout tətbiq etmək üçün xüsusi bir hüceyrə yarada bilərsiniz. Lakin daha sadə bir yol var: Keras tərəfindən təmin olunan əksər rekurrent qatlar və hüceyrələr dropout və recurrent_dropout hiperparametrlərinə malikdir - dropout girişlərə tətbiq olunacaq dropout nisbətini, recurrent_dropout isə gizli vəziyyətlər üçün, zaman addımları arasında dropout nisbətini təyin edir. Beləliklə, RNN-də hər zaman addımında dropout tətbiq etmək üçün xüsusi hüceyrə yaratmağa ehtiyac yoxdur.

Bu texnikalar sayəsində qeyri-sabit qradientlər problemini yüngülləşdirə və RNN-i daha səmərəli öyrədə bilərsiniz. İndi isə qısa müddətli yaddaş (short-term memory) problemini necə həll edəcəyimizə baxaq.

Qısa Müddətli Yaddaş Probleminin Həlli

RNN-də məlumatların keçdiyi transformasiyalar səbəbindən hər zaman addımında bəzi məlumatlar itir. Bir müddət sonra RNN-in vəziyyəti demək olar ki, ilk girişlərin heç bir izini saxlamır. Bu, ciddi problem

Tövsiyə

Zaman seriyalarını proqnozlaşdırarkən, proqnozlarınızla birlikdə bəzi səhv çubuqları (error bars) göstərmək tez-tez faydalıdır. Bunun üçün bir yanaşma 11-ci fəsildə təqdim olunan MC dropout-dan istifadə etməkdir: təlim zamanı recurrent_dropout istifadə edin, sonra modeli model(X, training=True) şəklində çağıraraq proqnoz zamanı dropout-u aktiv saxlayın. Bunu bir neçə dəfə təkrarlayaraq bir-birindən bir qədər fərqli bir neçə proqnoz əldə edin, sonra isə hər zaman addımı üçün bu proqnozların ortalamasını (mean) və standart sapmasını (standard deviation) hesablayın.

yarada bilər. Məsələn, Dory adlı balığı uzun bir cümləni tərcümə etməyə çalışarkən təsəvvür edin; o, cümləni oxuyub bitirdikdə, onun necə başladığını xatırlamır.

Bu problemi həll etmək üçün uzunmüddətli yaddaşa malik müxtəlif növ hüceyrələr təqdim olunub. Onlar o qədər uğurlu olmuşdur ki, əsas hüceyrələr artıq çox istifadə edilmir. Gəlin əvvəlcə bu uzunmüddətli yaddaş hüceyrələrinin ən populyarına baxaq: LSTM hüceyrəsi.

LSTM Hüceyrələri

Uzun-qısa müddətli yaddaş (Long Short-Term Memory, LSTM) hüceyrəsi 1997-ci ildə Sepp Hochreiter və Jürgen Schmidhuber tərəfindən təklif olunmuş və sonrakı illərdə Alex Graves, Haşim Sak və Wojciech Zaremba kimi tədqiqatçılar tərəfindən tədricən təkmilləşdirilmişdir.

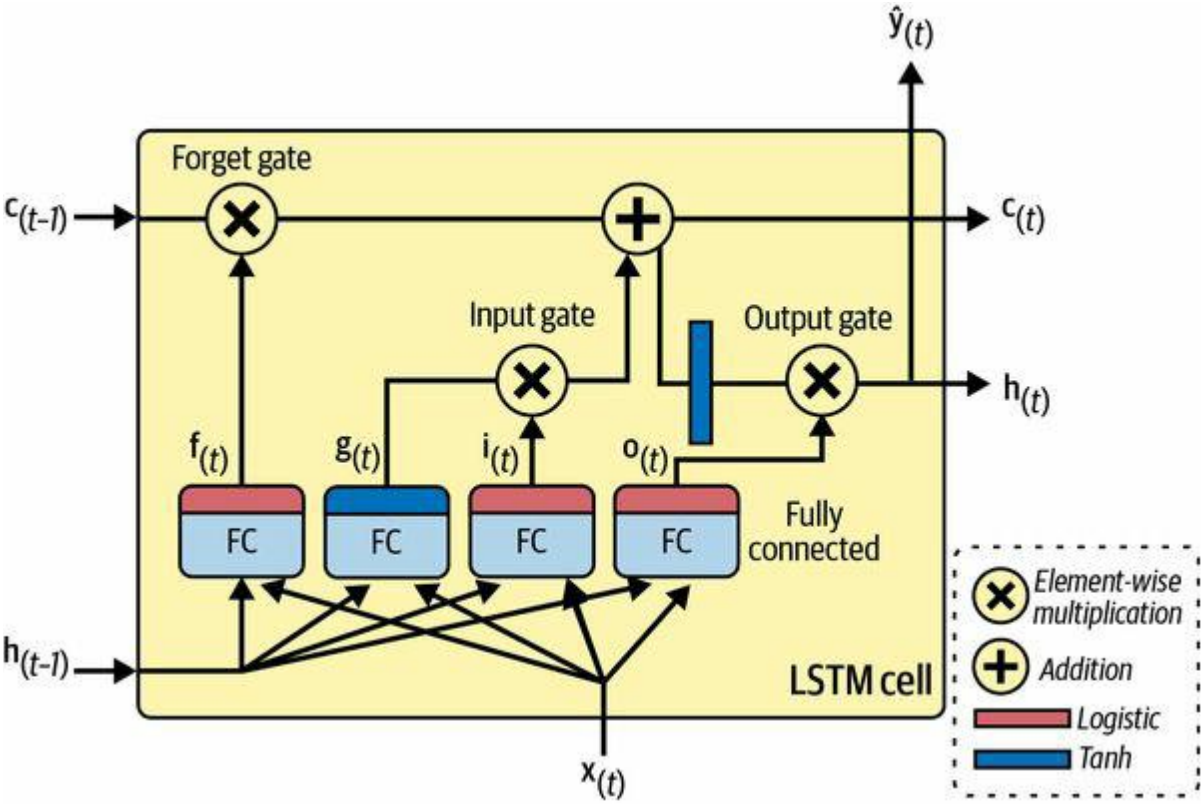
Əgər LSTM hüceyrəsini bir qara qutu (black box) kimi qəbul etsək, onu əsas hüceyrə kimi istifadə etmək mümkündür, lakin nəticələr xeyli yaxşı olacaq; təlim daha sürətli yaxınlaşacaq və məlumatlarda uzunmüddətli nümunələri aşkar edə biləcək.

Keras-da sadəcə SimpleRNN qatının yerinə LSTM qatından istifadə edə bilərsiniz.

```
model = tf.keras.Sequential([
    tf.keras.layers.LSTM(32, return_sequences=True, input_shape=[None, 5]),
    tf.keras.layers.Dense(14)
])
```

Alternativ olaraq, tf.keras.layers.RNN qatından istifadə edə bilərsiniz və ona arqument kimi bir LSTMCell verə bilərsiniz. Lakin LSTM qat GPU-da işləyərkən optimallaşdırılmış implementasiya istifadə edir (baxın: 19-cu fəsil), buna görə ümumiyyətlə onu istifadə etmək daha məqsədəuyğundur. (RNN qatı əsasən xüsusi hüceyrələr yaratdığınız zaman faydalıdır, necə ki, əvvəl göstərdik.)

Bəs LSTM hüceyrəsi necə işləyir? Onun arxitekturası Şəkil 15-12-də göstərilmişdir. Qutunun içində nə olduğunu nəzərə almasaq, LSTM hüceyrəsi adi hüceyrəyə çox bənzəyir, lakin onun vəziyyəti iki vektora bölünür: h və c (burada “c” “cell” mənasını verir). Siz h-ni qısa müddətli vəziyyət (short-term state), c-ni isə uzunmüddətli vəziyyət (long-term state) kimi düşünə bilərsiniz.



Təsvir 15-12. LSTM hüceyrəsi

İndi qutunu açaq! Əsas fikir ondadır ki, şəbəkə uzunmüddətli vəziyyətdə nəyi saxlamağı, nəyi atmağı və nəyi oxumağı öyrənə bilər.Uzunmüddətli vəziyyət $\mathbf{c}_{(t-1)}$ şəbəkədə soldan sağa hərəkət edərkən, əvvəlcə forget gate-dən keçir və bəzi xatirələri silir, sonra isə əlavə əməliyyatı vasitəsilə (addition operation) yeni xatirələr əlavə olunur (bu əməliyyat giriş qapısı (input gate) tərəfindən seçilmiş xatirələri əlavə edir). Nəticə olan $\mathbf{c}_{(t)}$ birbaşa çıxışa göndərilir, əlavə transformasiyaya ehtiyac yoxdur. Beləliklə, hər zaman addımında bəzi xatirələr silinir və bəzi xatirələr əlavə olunur.

Əlavə əməliyyatından sonra uzunmüddətli vəziyyət kopyalanır, tanh funksiyasından keçirilir və sonra output gate tərəfindən süzülür. Bu proses qısa müddətli vəziyyət $\mathbf{h}_{(t)}$-ni yaradır (bu da həmin zaman addımı üçün hüceyrənin çıxışına, $\mathbf{y}_{(t)}$, bərabərdir).

İndi gəlin qapıların necə işlədiyinə, həmçinin, yeni xatirələrin haradan gəldiyinə baxaq.

Əvvəlcə cari giriş vektoru $\mathbf{x}_{(t)}$ və əvvəlki qısa müddətli vəziyyət $\mathbf{h}_{(t-1)}$ dörd müxtəlif tam əlaqəli (fully connected) qata verilir. Onların hər biri fərqli məqsədə xidmət edir:

- Əsas qat $\mathbf{g}_{(t)}$ çıxışını verən qatdır. Onun adi rolu cari girişləri $\mathbf{x}_{(t)}$ və əvvəlki (qısa müddətli) vəziyyəti $\mathbf{h}_{(t-1)}$ analiz etməkdir.Sadə bir hüceyrədə bu qatdan başqa heç nə yoxdur və onun çıxışı birbaşa $\mathbf{y}_{(t)}$ və $\mathbf{h}_{(t)}$-yə ötürülür. Lakin LSTM hüceyrəsində bu qatın çıxışı birbaşa ötürülmür; əksinə, onun ən vacib hissələri uzun müddətli vəziyyətdə saxlanılır (qalan hissə isə atılır).

- Digər üç qat isə *qapı nəzarətçiləridir*(gate controllers). Onlar loqistik aktivləşdirmə funksiyasından istifadə etdiyinə görə çıxışları 0-dan 1-ə qədər dəyişir. Göründüyü kimi, qapı nəzarətçilərinin çıxışları element-wise vurma əməliyyatına verilir: əgər 0 çıxış verirlərsə qapı bağlı olur, əgər 1 çıxış verirlərsə qapı açılır. Xüsusilə:

- *Forget gate* (**$f_{(t)}$** ilə idarə olunur) uzun müddətli vəziyyətin hansı hissələrinin silinəcəyini idarə edir.
- *Input gate* (**$i_{(t)}$** ilə idarə olunur) **$g_{(t)}$**-nin hansı hissələrinin uzun müddətli vəziyyətə əlavə olunacağını idarə edir.
- *Output gate* (**$o_{(t)}$** ilə idarə olunur) isə uzun müddətli vəziyyətin hansı hissələrinin bu zaman addımında oxunacağını və həm **$h_{(t)}$**, həm də **$y_{(t)}$**-yə çıxış veriləcəyini idarə edir.

Qısa desək, LSTM hüceyrəsi əhəmiyyətli bir girişi tanımağı öyrənə bilər (bu, input gate-in roludur), onu uzun müddətli vəziyyətdə saxlaya bilər, lazım olduqca qoruyub saxlayar (bu, forget gate-in roludur) və lazım olanda çıxara bilər. Bu, həmin hüceyrələrin - zaman seriyaları, uzun mətnlər, audio qeydləri və digər məlumatlarda uzunmüddətli nümunələri tutmaqda niyə bu qədər uğurlu olduğunu izah edir.

Tənlik 15-4 hüceyrənin uzun müddətli vəziyyətini, qısa müddətli vəziyyətini və hər zaman addımında çıxışını tək bir nümunə üçün necə hesablamalı olduğunu ümumiləşdirir (bütün mini-batch üçün tənliklər çox oxşardır).

Tənlik 15-4. LSTM hesablamaları

$$
\begin{aligned}
i(t) &= \sigma(W_{xi}^{\top} x(t) + W_{hi}^{\top} h(t-1) + b_i) \\
f(t) &= \sigma(W_{xf}^{\top} x(t) + W_{hf}^{\top} h(t-1) + b_f) \\
o(t) &= \sigma(W_{xo}^{\top} x(t) + W_{ho}^{\top} h(t-1) + b_o) \\
g(t) &= \tanh(W_{xg}^{\top} x(t) + W_{hg}^{\top} h(t-1) + b_g) \\
c(t) &= f(t) \otimes c(t-1) + i(t) \otimes g(t) \\
y(t) &= h(t) = o(t) \otimes \tanh(c(t))
\end{aligned}
$$

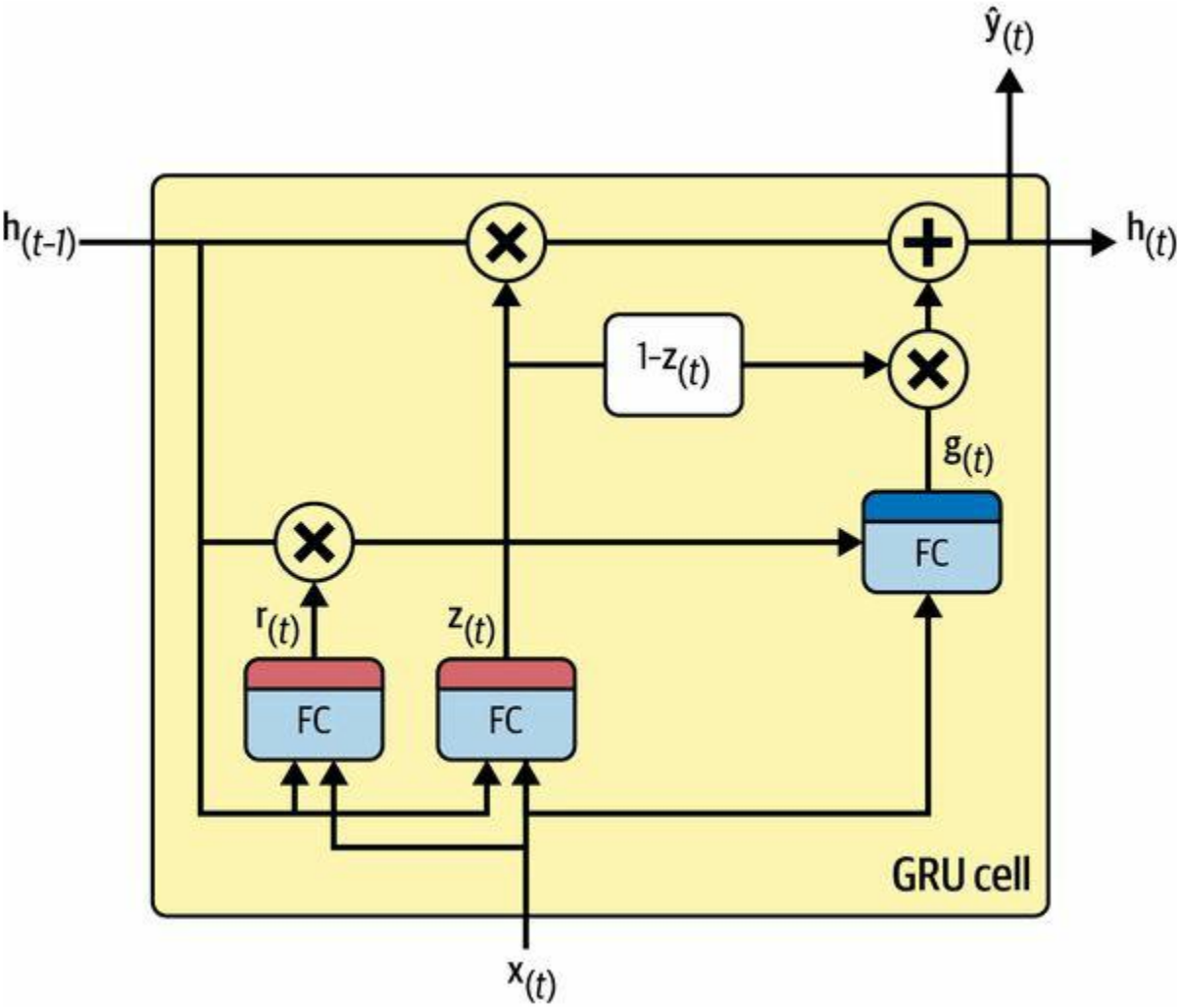
Bu tənlikdə:

- **$W_{(xi)}$, $W_{(xf)}$, $W_{(xo)}$ və $W_{(xg)}$** hər dörd qatın giriş vektoru **$x_{(t)}$** ilə əlaqəsi üçün çəkilər matrisləridir.
- **$W_{(hi)}$, $W_{(hf)}$, $W_{(ho)}$ və $W_{(hg)}$** hər dörd qatın əvvəlki qısa müddətli vəziyyət **$h_{(t-1)}$** ilə əlaqəsi üçün çəkilər matrisləridir.
- **b_i ,b_f, b_o və b_g** hər dörd qatın **bias** terminləridir. Qeyd etmək lazımdır ki, TensorFlow **b_f**-ni sıfır əvəzinə 1-lərlə dolu vektor kimi başladır. Bu, təlimin başlanğıcında hər şeyi unutmağın qarşısını alır.

LSTM hüceyrəsinin bir neçə fərqli variantı mövcuddur. Xüsusilə populyar olan variantlardan biri **GRU hüceyrəsidir**, ona indi baxacağıq.

GRU hüceyrələri

Gated Recurrent Unit (GRU) hüceyrəsi (bax: Təsvir 15-13) 2014-cü ildə Kyunghyun Cho və başqaları tərəfindən təklif olunmuşdur. Bu işdə əvvəlcə müzakirə etdiyimiz **encoder–decoder** şəbəkəsi də təqdim edilmişdir.



Təsvir 15-13 GRU hüceyrəsi

GRU hüceyrəsi LSTM hüceyrəsinin sadələşdirilmiş versiyasıdır eyni dərəcədə yaxşı nəticə göstərir (bu da onun getdikcə populyarlaşmasını izah edir). Əsas sadələşdirmələr bunlardır:

- Hər iki vəziyyət vektoru tək bir $\mathbf{h}_{(t)}$ vektorunda birləşdirilir.
- Tək qapı nəzarətçisi $\mathbf{z}_{(t)}$ həm unutma qapısını, həm də giriş qapısını idarə edir. Əgər qapı nəzarətçisi çıxışda 1 verirsə, unutma qapısı açıq olur (=1) və giriş qapısı bağlanır (1−1=0). Əgər çıxış 0 -dırsa, bunun əksi baş verir. Başqa sözlə, hər dəfə yaddaşa yeni məlumat yazılmalı olduqda, əvvəlcə həmin yaddaş hüceyrəsi silinir. Bu isə, əslində, LSTM hüceyrəsinin tez-tez istifadə edilən bir variantıdır.

- Burada çıxış qapısı yoxdur; hər zaman addımında tam vəziyyət vektoru çıxışa ötürülür. Bununla belə, yeni bir qapı nəzarətçisi $\mathbf{r}_{(t)}$ mövcuddur və o, əvvəlki vəziyyətin hansı hissəsinin əsas lay ($\mathbf{g}_{(t)}$) üçün göstəriləcəyini idarə edir.

Tənlik 15-5 hər bir nümunə üçün hüceyrənin vəziyyətini hər zaman addımında necə hesablamağı ümumiləşdirir.

Tənlik 15-5. GRU hesablama qaydaları

$$z(t) = \sigma(W_{xz}^{\intercal} x(t) + W_{hz}^{\intercal} h(t-1) + b_z) \quad r(t) = \sigma(W_{xr}^{\intercal} x(t) + W_{hr}^{\intercal} h(t-1) + b_r)$$

$$g(t) = \tanh(W_{xg}^{\intercal} x(t) + W_{hg}^{\intercal} (r(t) \otimes h(t-1)) + b_g$$

$$h(t) = z(t) \otimes h(t-1) + (1 - z(t)) \otimes g(t)$$

Keras tf.keras.layers.GRU qatını təqdim edir: onu istifadə etmək sadəcə SimpleRNN və ya LSTM qatını GRU ilə əvəz etməklə mümkündür. Bundan əlavə, Keras tf.keras.layers.GRUCell də təqdim edir; əgər GRU hüceyrəsinə əsaslanan xüsusi bir hüceyrə yaratmaq istəsəniz, ondan istifadə edə bilərsiniz.

LSTM və GRU hüceyrələri RNN-lərin uğurunun əsas səbəblərindən biridir. Yenə də, onlar sadə RNN-lərdən daha uzun ardıcıllıqlarla işləyə bilsələr də, hələ də nisbətən məhdud qısa müddətli yaddaşa sahibdirlər və 100 və ya daha çox zaman addımına malik ardıcıllıqlarda, məsələn, audio nümunələri, uzun zaman seriyaları və ya uzun cümlələrdə uzunmüddətli naxışları öyrənməkdə çətinlik çəkirlər. Bunun həlli yollarından biri giriş ardıcıllıqlarını qısaltmaqdır; məsələn, 1D konvolyusiya qatlarından istifadə etməklə.

Ardıcıllıqları işləmək üçün 1D konvolyusiya qatlarından istifadə
14-cü fəsildə gördüyümüz kimi, 2D konvolyusiya qatı şəkil üzərində bir neçə nisbətən kiçik kernelin (və ya filtrin) sürüşdürülməsi ilə işləyir və hər kernel üçün bir 2D xüsusiyyət xəritəsi çıxarır. Oxşar şəkildə, 1D konvolyusiya qatı bir neçə kerneli ardıcıllıq üzərində sürüşdürür və hər kernel üçün 1D xüsusiyyət xəritəsi yaradır.

Hər kernel çox qısa bir ardıcıllıq nümunəsini öyrənir (kernel ölçüsündən uzun olmayacaq). Məsələn, 10 kernel istifadə etsəniz, qatın çıxışı 10 ədəd 1D ardıcıllıqdan ibarət olacaq (hamısı eyni uzunluqda) və ya bu çıxışı tək 10D ardıcıllıq kimi də görə bilərsiniz.

Bu o deməkdir ki, təkrar olunan qatlar və 1D konvolyusiya qatlarından (və ya hətta 1D pooling qatlarından) ibarət qarışıq bir neyron şəbəkəsi qura bilərsiniz.

Əgər stride = 1 və "same" padding ilə 1D konvolyusiya qatı istifadə etsəniz, çıxış ardıcıllığının uzunluğu giriş ardıcıllığı ilə eyni olacaq. Lakin "valid" padding və ya stride > 1 istifadə edilsə, çıxış ardıcıllığı girişdən qısa olacaq, buna görə hədəfləri uyğunlaşdırmağı unutmayın.

Məsələn, aşağıdakı model əvvəlki model ilə eynidir, yalnız fərqi ondadır ki, giriş ardıcıllığını 2 dəfə azaldan (stride = 2 istifadə edərək) 1D konvolyusiya qatından başlayır. Kernel ölçüsü stride-dan böyükdür, buna görə bütün girişlər qatın çıxışını hesablamaq üçün istifadə olunacaq və nəticədə model yalnız əhəmiyyətsiz detalları ataraq faydalı məlumatı qorumağı öyrənə bilər.

Ardıcıllıqları qısaltmaq konvolyusiya qatının GRU qatlarının daha uzun naxışları aşkar etməsinə kömək edə bilər, buna görə giriş ardıcıllığının uzunluğunu 112 günə qədər iki dəfə artırmaq mümkündür.

Qeyd etmək lazımdır ki, hədəflərdə ilk üç zaman addımını da çıxartmalıyıq: kernel ölçüsü 4-dür, ona görə konvolyusiya qatının ilk çıxışı girişdəki 0–3 zaman addımlarına əsaslanacaq və ilk proqnozlar 1–14 zaman addımlarının əvəzinə 4–17 zaman addımları üçün veriləcək.Bundan əlavə, stride səbəbindən hədəfləri də 2 dəfə azaldaraq (downsample) uyğunlaşdırmalıyıq.

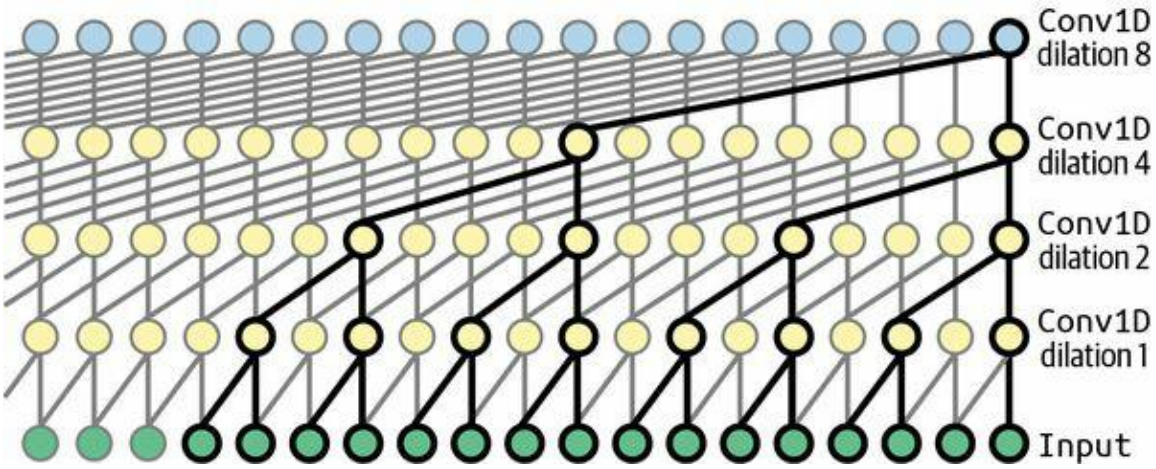
```
conv_rnn_model = tf.keras.Sequential([
    tf.keras.layers.Conv1D(filters=32, kernel_size=4, strides=2,
                           activation="relu", input_shape=[None, 5]),
    tf.keras.layers.GRU(32, return_sequences=True),
    tf.keras.layers.Dense(14)
])

longer_train = to_seq2seq_dataset(mulvar_train, seq_length=112,
                                  shuffle=True, seed=42)
longer_valid = to_seq2seq_dataset(mulvar_valid, seq_length=112)
downsampled_train = longer_train.map(lambda X, Y: (X, Y[:, 3::2]))
downsampled_valid = longer_valid.map(lambda X, Y: (X, Y[:, 3::2]))
[...] # compile and fit the model using the downsampled datasets
```

Əgər bu modeli təlim etsəniz və qiymətləndirsəniz, görəcəksiniz ki, o əvvəlki modeldən (kiçik bir fərqlə) daha yaxşı nəticə verir. Əslində, tamamilə yalnız 1D konvolyusiya qatlarından istifadə edib təkrar olunan qatları tamamilə çıxarmaq da mümkündür!

WaveNet

2016-cı ildə Aaron van den Oord və digər DeepMind tədqiqatçıları WaveNet adlanan yeni bir arxitektura təqdim etdilər. Onlar 1D konvolyusiya qatlarını yığıblar və hər qatda dilatasiya dərəcəsini iki dəfə artırıblar (hər neyronun girişlərinin nə qədər uzaqda olduğunu göstərir):İlk konvolyusiya qatı eyni anda yalnız iki zaman addımına baxır,növbəti qat dörd zaman addımını görür (onun qəbul sahəsi dörd zaman addımı uzunluqdadır),sonrakı qat səkkiz zaman addımını görür və s.

Beləliklə, aşağı qatlar qısa müddətli naxışları öyrənir, yuxarı qatlar isə uzunmüddətli naxışları öyrənir. dilation dərəcəsinin ikiqat artırılması sayəsində şəbəkə çox böyük ardıcıllıqları çox səmərəli şəkildə emal edə bilir.



Təsvir 15-14. WaveNet arxitekturası

Məqalənin müəllifləri əslində 1, 2, 4, 8, …, 256, 512 dilation dərəcələri ilə 10 konvolyusiya qatını yığıblar, sonra eyni dilation dərəcələri ilə başqa bir 10 qatlıq eyni qrup əlavə ediblər və daha sonra yenidən başqa bir 10 qatlıq eyni qrup yığıblar. Onlar bu arxitekturanı belə əsaslandırıblar ki, bu dilation dərəcələrinə malik tək bir 10 qatlıq konvolyusiya yığımı 1,024 ölçülü kernelə bərabər çox səmərəli bir konvolyusiya qatı kimi işləyəcək (amma daha sürətli, daha güclü və əhəmiyyətli dərəcədə az parametr istifadə edərək).

Həmçinin, şəbəkə boyunca ardıcıllıq uzunluğunu qorumaq üçün hər qatdan əvvəl giriş ardıcıllıqlarını dilation dərəcəsinə bərabər sayda sıfır ilə sol tərəfdən doldurublar.Aşağıda isə əvvəlki ardıcıllıqları eyni şəkildə işləmək üçün sadələşdirilmiş WaveNet-in necə tətbiq olunacağı göstərilir.

```
wavenet_model = tf.keras.Sequential()
wavenet_model.add(tf.keras.layers.Input(shape=[None, 5]))
for rate in (1, 2, 4, 8) * 2:
    wavenet_model.add(tf.keras.layers.Conv1D(
        filters=32, kernel_size=2, padding="causal", activation="relu",
        dilation_rate=rate))
wavenet_model.add(tf.keras.layers.Conv1D(filters=14, kernel_size=1))
```

Bu Sequential model açıq bir giriş qatı ilə başlayır—bu, yalnız ilk qat üçün input_shape təyin etməyə çalışmaqdan daha sadədir. Daha sonra "causal" padding istifadə edən 1D konvolyusiya qatı əlavə olunur; bu, "same" padding-ə bənzəyir, amma sıfırlar yalnız giriş ardıcıllığının əvvəlinə əlavə olunur, hər iki tərəfə deyil. Bu, konvolyusiya qatının proqnoz verərkən gələcəyə baxmamasını təmin edir.

Sonra, böyüyən dilation dərəcələri (1, 2, 4, 8 və yenidən 1, 2, 4, 8) ilə oxşar qat cütləri əlavə olunur. Nəhayət, çıxış qatı əlavə edilir: 14 filter ölçülü, ölçüsü 1 olan və heç bir aktivasiya funksiyası olmayan konvolyusiya qatı. Daha əvvəl gördüyümüz kimi, belə bir konvolyusiya qatı 14 vahidli Dense qatına bərabərdir.Causal padding sayəsində hər bir konvolyusiya qatı giriş ardıcıllığı ilə eyni uzunluqda çıxış verir, buna görə təlim zamanı istifadə etdiyimiz hədəflər tam 112 günlük ardıcıllıqlar ola bilər; onları kəsməyə və ya azaltmağa ehtiyac yoxdur.Bu bölümdə müzakirə etdiyimiz modellər nəqliyyat sərnişinliyi

proqnozlaşdırma tapşırığı üçün oxşar performans göstərir, amma tapşırıq növünə və mövcud məlumatın miqdarına görə əhəmiyyətli dərəcədə fərqlənə bilər. WaveNet məqaləsində müəlliflər müxtəlif audio

XƏBƏRDARLIQ

Əgər ən yaxşı Çikaqo sərnişinlik modellərimizi 2020-ci ildən başlayan test dövründə qiymətləndirsəniz, görəcəksiniz ki, onlar gözlənildiyi qədər yaxşı nəticə vermir! Bunun səbəbi nədir? Belə ki, həmin vaxt Covid-19 pandemiyası başladı və bu, ictimai nəqliyyata ciddi təsir göstərdi.

Daha əvvəl qeyd edildiyi kimi, bu modellər yalnız keçmişdən öyrəndikləri naxışlar gələcəkdə də davam edərsə yaxşı işləyəcək. Hər halda, bir modeli istehsalata yerləşdirməzdən əvvəl onun son məlumatlarla yaxşı işlədiyini yoxlayın.Və istehsalata yerləşdirildikdən sonra performansını müntəzəm olaraq izləməyi unutmayın.

tapşırıqlarda (bu arxitekturanın adının səbəbi də budur) müasir dövrün ən yaxşı performansını əldə ediblər, o cümlədən mətn-dan-səs (text-to-speech) tapşırıqlarında, bir neçə dildə son dərəcə realist səslər istehsal ediblər. Onlar həmçinin modeli musiqi yaratmaq üçün istifadə ediblər, hər dəfə bir audio nümunəsi istehsal edərək. Bu nailiyyət daha da təsir edicidir, çünki bir saniyəlik audio minlərlə zaman addımı ehtiva edə bilər—hətta LSTM və GRU-lar belə buzun ardıcıllıqları idarə edə bilmir.

Bununla artıq hər cür zaman sıralarını (time series) işləyə bilərsiniz! 16-cı fəsildə biz RNN-ləri araşdırmağa davam edəcəyik və onların müxtəlif NLP tapşırıqlarını, o cümlədən tərcümə etməyi necə həll edə bildiyini görəcəyik.

Tapşırıqlar

1. Sequence-to-sequence RNN üçün bir neçə tətbiq nümunəsi düşünə bilərsinizmi? Bəs sequence-to-vector RNN və vector-to-sequence RNN üçün necə?
2. RNN qatının girişləri neçə ölçüyə malik olmalıdır? Hər ölçü nəyi ifadə edir? Bəs onun çıxışları necə?
3. Əgər siz dərin sequence-to-sequence RNN qurmaq istəyirsinizsə, hansı RNN qatlarında return_sequences=True olmalıdır? Bəs sequence-to-vector RNN üçün necə?
4. Gündəlik univariat zaman sırasına (time series) malik olduğunuzu və növbəti yeddi günü proqnozlaşdırmaq istədiyinizi fərz edin. Hansı RNN memarlığından istifadə etməlisiniz?
5. RNN-ləri öyrətməyin əsas çətinlikləri hansılardır? Bunlarla necə mübarizə aparmaq olar?
6. LSTM hüceyrəsinin memarlığını çəkə bilərsinizmi?
7. Niyə RNN-də 1D konvolyusiya qatlarından istifadə etmək istəyə bilərsiniz?
8. Videoları təsnif etmək üçün hansı neyron şəbəkə memarlığından istifadə edə bilərsiniz?
9. TensorFlow Datasets-də mövcud olan SketchRNN dataset-i üçün təsnifat modeli öyrədin.
10. Bach xoral datasetini yükləyin və unzip edin. O, Johann Sebastian Bach tərəfindən yazılmış 382 xoraldan ibarətdir. Hər xoral 100-dən 640-a qədər time step-dən ibarətdir və hər time step 4 tam ədəddən ibarətdir. Hər tam ədəd pianoda notun indeksinə uyğundur (dəyəri 0 olduqda isə heç bir not ifa olunmur). Xoralın time step-lərindən verilmiş bir ardıcıllığa baxaraq növbəti time step-i (dörd not) proqnozlaşdıra bilən bir model öyrədin — rekurrent, konvolyusional və ya hər ikisinin birləşməsi. Sonra bu modeli istifadə edərək Bach üslubunda musiqi yaradın: xoralın başlanğıc hissəsini modelə verin və ondan növbəti time step-i proqnozlaşdırmasını istəyin, sonra bu time step-i girişə əlavə edin və modeldən növbəti notu soruşun, və s. Həmçinin Google-un Bach haqqında hazırlanmış maraqlı doodle üçün istifadə etdiyi Coconet modelinə də baxmağı unutmayın.

Bu tapşırıqların həlləri fəsilin sonundakı notebook-da mövcuddur:
https://homl.info/colab3

Qeydlər

1. Bir çox tədqiqatçılar RNN-lərdə ReLU əvəzinə hiperbolik tangent (tanh) aktivasiya funksiyasına üstünlük verirlər. Məsələn, Vu Pham və digərlərinin 2013-cü ildəki "*Dropout Improves Recurrent Neural Networks for Handwriting Recognition*" məqaləsinə baxın. ReLU-əsaslı RNN-lər də mümkündür, Quoc V. Le və digərlərinin 2015-ci ildəki "*A Simple Way to Initialize Recurrent Networks of Rectified Linear Units*" məqaləsində göstərildiyi kimi.
2. Nal Kalchbrenner və Phil Blunsom, "*Recurrent Continuous Translation Models*", Empirical Methods in Natural Language Processing Konfransı (2013): 1700–1709.
3. Chicago Transit Authority-yə aid ən son məlumatlar Chicago Data Portal-da mövcuddur.
4. Yaxşı hiperparametrlərin seçilməsi üçün daha prinsipial yanaşmalar da mövcuddur — məsələn, autokorrelyasiya funksiyası (ACF) və qismən autokorrelyasiya funksiyasının (PACF) təhlili, yaxud çox parametrli modelləri cəzalandırmaq və overfitting riskini azaltmaq üçün AIC və BIC metriklərinin istifadəsi. Amma grid search başlanğıc üçün yaxşı üsuldur. Daha ətraflı məlumat üçün Jason Brownlee-nin ACF-PACF mövzusundakı yazısına baxa bilərsiniz.
5. Yoxlama dövrü 1 yanvar 2019-da başlayır, buna görə də ilk proqnoz 26 fevral 2019 üçündür (səkkiz həftə sonra). Biz baza modellərini qiymətləndirərkən proqnozlara 1 martdan başlamışdıq, amma bu da kifayət qədər yaxın olacaq.
6. Bu model üzərində sərbəst şəkildə təcrübə edə bilərsiniz. Məsələn, növbəti 14 gün üçün həm avtobus, həm də metro sərnişinlərini proqnozlaşdırmağa çalışa bilərsiniz. Bunun üçün hədəfləri hər ikisini əhatə edəcək şəkildə dəyişməli və modelin çıxışlarını 14 yox, 28 proqnoz verməsi üçün tənzimləməlisiniz.
7. César Laurent və digərləri, "*Batch Normalized Recurrent Neural Networks*", IEEE ICASSP (2016): 2657–2661.
8. Jimmy Lei Ba və digərləri, "*Layer Normalization*", arXiv preprint arXiv:1607.06450 (2016).
9. Daha sadə yanaşma SimpleRNNCell-dən miras almaq olardı, beləliklə daxili SimpleRNNCell yaratmağa və state_size və output_size atributlarını idarə etməyə ehtiyac qalmazdı. Amma məqsəd sıfırdan xüsusi (custom) hüceyrənin necə yaradıldığını göstərmək idi.
10. "Finding Nemo" və "Finding Dory" animasiya filmlərində qısamüddətli yaddaş itkisi olan bir personaj.

11. Sepp Hochreiter və Jürgen Schmidhuber, "*Long Short-Term Memory*", Neural Computation 9, № 8 (1997): 1735–1780.
12. Haşim Sak və digərləri, "*Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition*", arXiv preprint arXiv:1402.1128 (2014).
13. Wojciech Zaremba və digərləri, "*Recurrent Neural Network Regularization*", arXiv preprint arXiv:1409.2329 (2014).
14. Kyunghyun Cho və digərləri, "*Learning Phrase Representations Using RNN Encoder–Decoder for Statistical Machine Translation*", EMNLP (2014): 1724–1734.
15. Klaus Greff və digərləri, "*LSTM: A Search Space Odyssey*", IEEE Transactions on Neural Networks and Learning Systems 28, № 10 (2017): 2222–2232. Bu məqalə göstərir ki, bütün LSTM variantları təxminən eyni performans göstərir.
16. Aaron van den Oord və digərləri, "*WaveNet: A Generative Model for Raw Audio*", arXiv preprint arXiv:1609.03499 (2016).
17. Tam WaveNet modeli bir neçə əlavə texnikadan istifadə edir, məsələn ResNet-də olduğu kimi skip connection-lar və GRU hüceyrəsindəki kimi gated aktivasiya vahidləri. Daha ətraflı məlumat üçün bu fəsilin notebook-una baxın.