10-cu Fəsil

Keras ilə Süni Sinir Şəbəkələrinə Giriş

Quşlar bizə uçmağa, Pıtraq otu (Burdock) isə Velkronu ixtira etməyə ilham verdi və təbiət saysız-hesabsız başqa texnoloji kəşflərə yol açdı. Belə olduqda, ağıllı maşın yaratmaq üçün beynimizin arxitekturasından ilham almaq tamamilə məntiqli görünür. Məhz bu məntiq *Süni Sinir Şəbəkələrinin (Artificial Neural Networks – ANN)* meydana gəlməsinə gətirib çıxardı: ANN – beynimizdəki bioloji neyron şəbəkələrindən ilhamlanmış bir Maşın Öyrənməsi(machine learning) modelidir.

Necə ki, təyyarələr quşlardan ilham alsa da qanad çırpmaq məcburiyyətində deyillər, ANN-lər də bioloji analoqlarından getdikcə çox uzaqlaşmağa başlayıblar. Bəzi tədqiqatçılar hətta "bioloji bənzətməni tamamilə buraxmaq" lazım olduğunu deyirlər (məsələn, "neyron" əvəzinə "vahid" ifadəsini işlətməklə), çünki əks halda yaradıcılığımızı yalnız bioloji baxımdan mümkün olan sistemlərlə məhdudlaşdıra bilərik.[1]

ANN-lər Dərin Öyrənmənin (Deep Learning) əsasını təşkil edir. Onlar çox yönlü, güclü və genişləndirilə bilən modellərdir. Bu səbəbdən böyük və çox mürəkkəb Maşın Öyrənməsi məsələlərini həll etmək üçün idealdır məsələn, milyardlarla şəkillərin təsnifatı (məs:Google Images),nitqin tanınması xidmətləri (məs:Apple Siri),hər gün yüz milyonlarla istifadəçiyə video tövsiyə etmək (məs:YouTube)və ya Go oyununda dünya çempionunu məğlub etməyi öyrənmək (məs:DeepMind-in AlphaGo-su).

Bu fəsilin ilk hissəsi süni neyron şəbəkələrinin əsaslarını izah edir. Əvvəlcə ən sadə ANN arxitekturalarına baxacağıq, sonra isə bu gün ən çox istifadə olunan *Çoxqatlı Perseptronlara (Multilayer Perceptrons – MLP)* keçid edəcəyik (digər arxitekturaları isə növbəti fəsillərdə öyrənəcəyik). İkinci hissədə məşhur *Keras API(Application Programming Interface-Tətbiq proqramlaşdırma İnterfeysi)*-si ilə neyron şəbəkələrinin necə tətbiq olunacağını öyrənəcəyik. Keras neyron şəbəkələrinin qurulması, öyrədilməsi, qiymətləndirilməsi və işə salınması üçün gözəl dizayn edilmiş və sadə, yüksək səviyyəli bir API-dir.

Amma sadəliyinə aldanmayın: o, geniş çeşidli neyron şəbəkə arxitekturaları qurmağa imkan verəcək qədər anlamlı və çevikdir. Əslində, çox güman ki, əksər istifadə halları üçün sizin üçün kifayət edəcəkdir.
Və əgər əlavə çeviklik lazım olsa, həmişə aşağı səviyyəli API-dən istifadə etməklə özəl Keras

[1] Əgər siz bioloji ilhamlara açıq olsanız, eyni zamanda işlədiyi müddətcə bioloji baxımdan qeyri-realist modellər yaratmaqdan çəkinməsəniz bu iki dünyadan daha yaxşı nəticələr əldə edə bilərsiniz.

komponentləri yaza bilərsiniz – bunu 12-ci fəsildə görəcəyik.
Amma əvvəlcə, süni neyron şəbəkələrinin necə yarandığını görmək üçün keçmişə nəzər salaq!

Bioloji Neyronlardan Süni Neyronlara

Təəccüblüdür ki, süni neyron şəbəkələri (ANN) artıq xeyli müddətdir mövcuddur: onlar ilk dəfə 1943-cü ildə neyrofizioloq Uorren MakKallox və riyaziyyatçı Volter Pitts tərəfindən təqdim edilmişdir. Onların *"Sinir Fəaliyyətinin İmmanent(öz təbiətinə xas) Olan İdeyaların Məntiqi Hesabı"* adlı məşhur məqaləsində[2] MakKallox və Pitts heyvan beynindəki bioloji neyronların hökm məntiqi (propositional logic) ilə mürəkkəb hesablama əməliyyatlarını aparmaq üçün necə birlikdə işləyə biləcəyini təmsil edən sadələşdirilmiş hesablama modelini təqdim etdilər. Bu, ilk süni neyron şəbəkəsi arxitekturası idi. O vaxtdan bəri bir çox digər arxitekturalar ixtira olunmuşdur və irəlidə bunları görəcəyik.

ANN-lərin ilkin nailiyyətləri bizim tezliklə həqiqətən ağıllı maşınlarla danışacağımıza dair geniş yayılmış bir inama səbəb oldu. 1960-cı illərdə bu vədin yerinə yetirilməyəcəyi aydın olduqda (ən azı uzun müddət), maliyyələşmə başqa sahələrə yönəldi və ANN-lər uzun bir qış dövrünə daxil oldular. 1980-ci illərin əvvəllərində yeni arxitekturalar ixtira edildi və daha yaxşı öyrətmə texnikaları inkişaf etdirildi, bu da konnektionizmə (neyron şəbəkələrinin öyrənilməsi - *connectionism)* marağın yenidən canlanmasına səbəb oldu. Lakin inkişaf ləng gedirdi və 1990-cı illərdə Dəstək Vektor Maşınları (Support Vector Machines, bax Fəsil 5-ə) kimi digər güclü Maşın Öyrənməsi(ML-Machine Learning) metodları yaradıldı. Bu texnikalar ANN-lərlə müqayisədə daha yaxşı nəticələr və daha güclü nəzəri əsaslar təklif edirmiş kimi göründü və nəticədə süni neyron şəbəkələrinin araşdırılması bir daha kənara qoyuldu.

Hal-hazırda biz ANN-lərə marağın yeni bir dalğasının şahidi oluruq. Bu dalğa əvvəlkilər kimi sönəcəkmi? Bu dəfə fərqli olacağına və ANN-lərə yenidən artan marağın həyatımıza daha dərin təsir göstərəcəyinə inanmaq üçün bir neçə əsaslı səbəb var:

- İndi neyron şəbəkələrinin öyrədilməsi üçün böyük həcmdə məlumat bolluğu var və ANN-lər çox vaxt böyük və mürəkkəb problemlərdə digər ML metodlarını üstələyirlər.
- 1990-cı illərdən bəri hesablama gücündə ciddi artım baş verib və bu, böyük neyron şəbəkələrinin məqbul hesab ediləcək müddətdə öyrədilməsini(training) mümkün edir. Bu, qismən Moore qanununa görədir (inteqrasiya olunmuş çərçivələrdə komponentlərin sayı son 50 ildə təxminən hər 2 ildən bir iki dəfə artıb), həmçinin də oyun sənayesi sayəsindədir, çünki o, milyonlarla güclü GPU kartlarının istehsalını stimullaşdırdı. Bundan əlavə, bulud platformaları bu gücü hamı üçün əlçatan etdi.

[2] Uorren S. MakKallox və Volter Pitts. "Sinir Fəaliyyətində İmmanent olan İdeyaların Məntiqi Hesabı." *Riyazi Biofizika Bülleteni*, cild 5, № 4 (1943): 115–133.

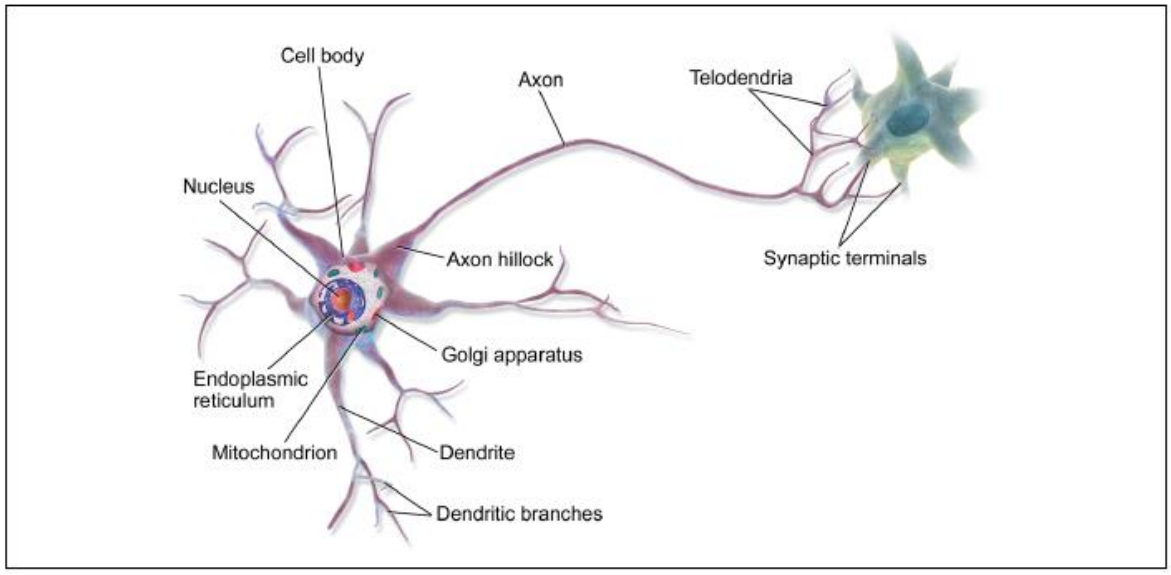
- Öyrətmə alqoritmlərinin təkmilləşmişdir. Düzünü desək, onlar 1990-cı illərdə istifadə edilənlərdən cüzi dərəcədə fərqlənir, amma bu kiçik dəyişikliklər belə çox böyük müsbət təsir göstərmişdir.
- ANN-in bəzi nəzəri məhdudiyyətləri praktikada təhlükə yaratmadı. Məsələn, bir çoxları ANN öyrətmə alqoritmlərinin lokal optimumlarda ilişib qalacağına görə uğursuz olacağını düşünürdü, amma praktikada bu çox nadir hallarda baş verir (və belə olduqda da, adətən qlobal optimuma olduqca yaxın nöqtədə olurlar).
- Belə görünür ki, ANN-lər maliyyələşmə və inkişaf baxımdan müsbət dövrə keçid ediblər. ANN-lər əsasında yaradılmış möhtəşəm məhsullar mütəmadi olaraq xəbər başlıqlarına çıxır, bu isə daha çox diqqət və maliyyə cəlb edir, nəticədə daha çox inkişaf və daha da təsirli məhsullar ortaya çıxır.

Bioloji Neyronlar

Süni neyronları müzakirə etməzdən əvvəl, gəlin qısa olaraq bioloji neyrona nəzər salaq (Şəkil 10-1-də göstərildiyi kimi). Bioloji neyron əsasən heyvan beyinlərində rast gəlinən, qeyri-adi görkəmli bir hüceyrədir. Onun tərkibinə nüvəni və hüceyrənin mürəkkəb komponentlərinin çoxunu özündə saxlayan *hüceyrə cismi,* çoxsaylı budaqlanmış çıxıntılar olan *dendritlər*, həmçinin *akson* adlanan çox uzun bir çıxıntı daxildir. Aksonun uzunluğu hüceyrə cisminin uzunluğundan ya bir neçə dəfə çox ola bilər və ya on minlərlə dəfə artıq ola bilər. Aksonun uc hissəsinə yaxın yerdə o, *telodendriya* adlanan çoxsaylı budaqlara ayrılır və bu budaqların ucunda *sinaptik terminallar* (və ya sadəcə *sinapslar*) yerləşir. Bu sinapslar digər neyronların dendritləri və ya hüceyrə cisimləri ilə əlaqələnir.[3]

Bioloji neyronlar *fəaliyyət potensialı* (AP, və ya sadəcə *siqnal*) adlanan qısa elektrik impulsları yaradırlar. Bu impulslar akson boyunca hərəkət edir və sinapsların *neyrotransmitterlər* adlanan kimyəvi siqnalları buraxmasına səbəb olur. Neyron çox qısa müddət ərzində (bir neçə millisaniyə ərzində) kifayət qədər neyrotransmitter qəbul etdikdə, öz elektrik impulsunu yaradır (əslində bu neyrotransmitterlərin növündən asılıdır, çünki onların bəziləri neyronun impuls yaratmasının qarşısını alır).

[3] Onlar əslində bir-birinə bağlı deyillər, sadəcə o qədər yaxın yerləşirlər ki, kimyəvi siqnalları çox sürətli şəkildə mübadilə edə bilirlər.

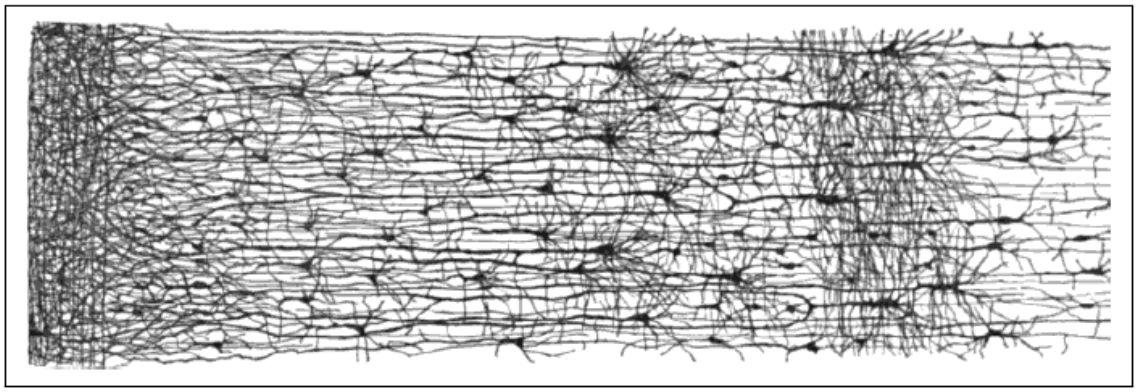


Şəkil 10-1. Bioloji neyron[4]

Beləliklə, ayrı-ayrı bioloji neyronların davranışı nisbətən sadə görünsə də, onlar milyardlarla neyronun təşkil etdiyi geniş bir şəbəkədə birləşmiş vəziyyətdədirlər və hər neyron adətən minlərlə başqa neyronla əlaqələnir. Sadə neyronlardan ibarət bir şəbəkə vasitəsilə yüksək dərəcədə mürəkkəb hesablama əməliyyatları yerinə yetirilə bilər. Bu proses, sadə qarışqaların birgə fəaliyyəti nəticəsində mürəkkəb qarışqa yuvasının yaranmasına bənzəyir.

Bioloji neyron şəbəkələrinin (BNN-lərin)[5] arxitekturası hələ də fəal tədqiqat mövzusudur. Lakin beynin bəzi hissələri artıq xəritələndirilmişdir və görünür ki, neyronlar çox vaxt ardıcıl qatlar şəklində təşkil olunurlar. Bu xüsusilə beyin qabığında (yəni beynin xarici qatı olan böyük yarımkürə qabığında) müşahidə olunur (Bax: şəkil 10-2.).

[4] Şəkil müəllifi: Bruce Blaus (Creative Commons 3.0). Mənbə: *https://en.wikipedia.org/wiki/Neuron*.
[5] Maşın öyrənməsi kontekstində "neyron şəbəkələr" ifadəsi, adətən, bioloji neyron şəbəkələrinə (BNN-lərə) deyil, süni neyron şəbəkələrinə (ANN-lərə) istinad edir.

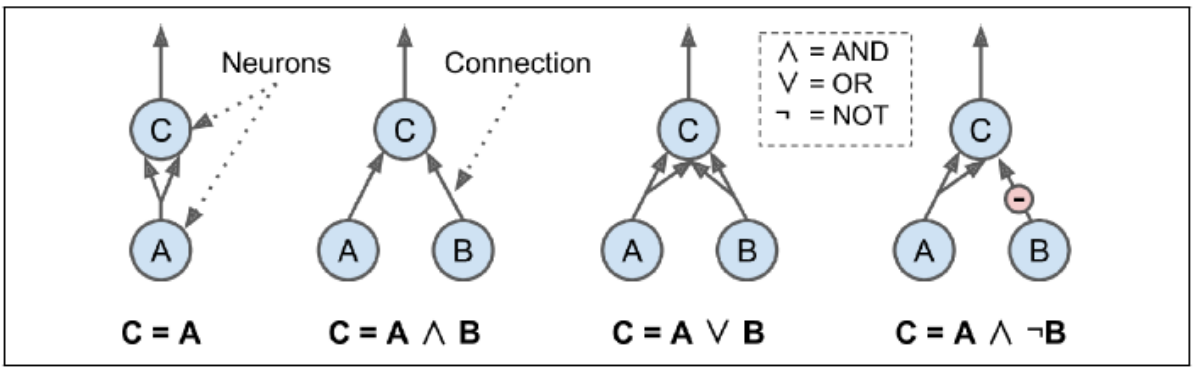


Şəkil 10-2. Bioloji neyron şəbəkəsində çoxsaylı qatlar (insan beyin qabığı)[6]

Neyronlarla Məntiqi Hesablama

MakKallox və Pitts bioloji neyronun çox sadə bir modelini təklif etdilər hansı ki, bu model sonradan *süni neyron* adı ilə tanındı. Bu modelin bir və ya bir neçə ikili (açıq/söndürülmüş) giriş verilənləri(input) və bir ikili çıxış nəticəsi(output) var. Süni neyronun outputu yalnız inputların müəyyən saydan daha çoxu aktiv olduqda işə düşür. Onlar öz məqalələrində göstərdilər ki, bu qədər sadələşdirilmiş modeldən istifadə etməklə belə, istənilən məntiqi hökmü hesablaya bilən neyron şəbəkəsi qurmaq mümkündür.

Bu cür şəbəkənin necə işlədiyini görmək üçün gəlin müxtəlif məntiqi hesablamalar aparan bir neçə süni neyron şəbəkəsi (Şəkil 10-3-ə bax) quraq və fərz edək ki, neyron onun ən azı iki inputu aktiv olduqda işə düşür.



Şəkil 10-3. Sadə məntiqi hesablama aparan süni neyron şəbəkələri (ANN-lər)

[6] Ramon y Cajal tərəfindən çəkilmiş kortikal laminasiya təsviri (ictimai mülkiyyət).
Mənbə: *https://en.wikipedia.org/wiki/Cerebral_cortex*.

Gəlin baxaq bu şəbəkələr nə edir:

- Soldakı birinci şəbəkə eynilik funksiyasıdır: əgər A neyronu aktivdirsə, C neyronu da aktiv olur (çünki C neyronu A-dan iki input siqnalı alır); əgər A neyronu söndürülmüşdürsə, C də söndürülür.
- İkinci şəbəkə məntiqi AND (VƏ) funksiyasını icra edir: C neyronu yalnız A və B neyronları eyni vaxtda aktiv olduqda işə düşür (tək bir input siqnalı C-ni aktivləşdirmək üçün kifayət deyil).
- Üçüncü şəbəkə məntiqi OR (VƏ YA) funksiyasını yerinə yetirir: C neyronu A və ya B neyronlarından hər hansı biri (və ya hər ikisi) aktiv olduqda işə düşür.
- Nəhayət, input bağlantılarının neyronun fəaliyyətini ləngidə biləcəyini fərz etsək (bioloji neyronlarda olduğu kimi) onda dördüncü şəbəkə bir qədər mürəkkəb məntiqi ifadəni hesablaya bilir. Bu halda, C neyronu yalnız A neyronu aktiv və B neyronu söndürülmüş olduqda işə düşür. Əgər A neyronu daim aktivdirsə, onda nəticədə **məntiqi NOT** funksiyası yaranır: B söndürülmüş olduqda C aktiv olur, əks halda isə əksinə.

Bu şəbəkələrin müxtəlif kombinasiyalarının mürəkkəb məntiqi ifadələri necə hesablaya biləcəyini təsəvvür edə bilərsiniz (bax: fəsil sonundakı tapşırıqlar).

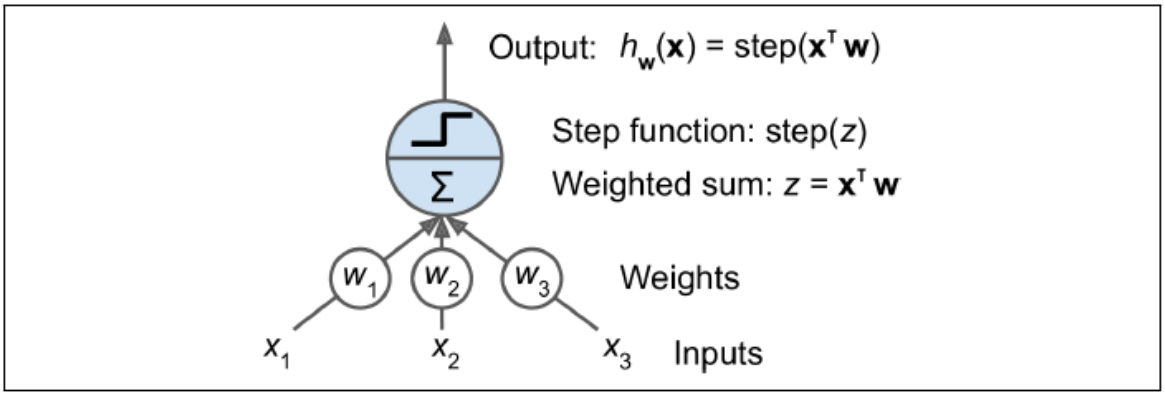
Perseptron (Perceptron)

Perseptron süni neyron şəbəkələrinin(ANN) ən sadə arxitekturalarından biridir və 1957-ci ildə Frank Rozenblatt tərəfindən ixtira edilmişdir. O, bir qədər fərqli süni neyrona əsaslanır (bax: Şəkil 10-4) və bu neyron *həd məntiqi vahidi* (Threshold Logic Unit – TLU) və ya bəzən *xətti həd vahidi* (Linear Threshold Unit – LTU) adlanır.

Bu modeldə inputlar(girişlər) və outputlar(çıxışlar) rəqəmlərdir(ikili açıq/söndürülmüş dəyərlər əvəzinə). Hər bir input bağlantısı müəyyən bir çəki (weight) ilə əlaqələndirilir. TLU öz inputlarının çəkili cəmini hesablayır:

$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = \boldsymbol{x}^T \boldsymbol{w}$

Daha sonra bu cəmlənmiş dəyərə *addım funksiyası* (step function) tətbiq olunur və nəticə output kimi verilir: $\boldsymbol{h_w}(\boldsymbol{x}) = step(z)$, burada z $= \boldsymbol{x}^T \boldsymbol{w}$



Şəkil 10-4. Hədd məntiqi vahidi (TLU):İnputların(Girişlərin) çəkili cəmini hesablayan və daha sonra addım funksiyasını tətbiq edən süni neyron.

Perseptronlarda ən çox istifadə olunan addım funksiyası *Heaviside addım funksiyasıdır* (bax: Tənlik 10-1). Bəzən isə onun əvəzinə işarə funksiyası (sign function) tətbiq olunur.

Tənlik 10-1. Perseptronlarda istifadə olunan ümumi addım funksiyaları (hədd = 0 olduğu halda):

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

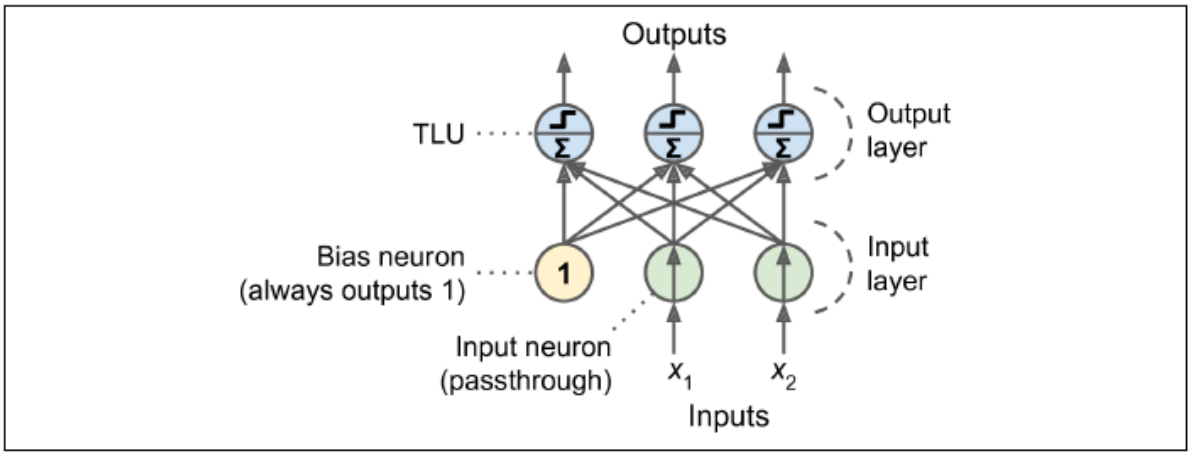
Sadə xətti ikili təsnifat (binary classification) üçün tək bir TLU istifadə oluna bilər. O, inputların xətti kombinasiyasını hesablayır və nəticə həddi aşarsa, output olaraq müsbət təsnifat verir, əks halda isə mənfi təsnifat verir (logistik reqressiya və ya xətti SVM təsnifatçısı ilə eyni prinsiplə).

Məsələn, tək bir TLU-dan istifadə etməklə iris güllərinin ləçək uzunluğu və eninə əsasən təsnifatı aparmaq olar (üstəlik, əvvəlki fəsillərdə olduğu kimi əlavə sürüşmə (bias) xüsusiyyəti $x_0 = 1$ əlavə olunur). Bu halda TLU-nun öyrədilməsi, uyğun çəki dəyərlərini w_0, w_1və w_2 üçün tapmaq deməkdir (təlim alqoritmi qısa şəkildə izah edilir).

Perseptron sadəcə olaraq bir təbəqə TLU-dan[7] ibarətdir və hər bir TLU bütün inputlara qoşulmuş olur. Əgər bir təbəqədəki bütün neyronlar əvvəlki təbəqədəki hər bir neyrona (yəni onun input neyronlarına) qoşulmuşdursa, bu təbəqə *tam bağlı təbəqə* (fully connected layer) və ya *sıx təbəqə* (dense layer) adlanır. Perseptronun inputları *input neyronları* adlandırılan xüsusi ötürücü neyronlara (passthrough neurons) verilir: onlara sadəcə olaraq hansi input yedirilirsə onu da output olaraq verirlər. Bütün bu input neyronları *input təbəqəsini* təşkil edir. Bundan əlavə, adətən əlavə sürüşmə xüsusiyyəti (bias feature) ($x_0 = 1$) əlavə olunur: bu, daim 1 outputunu verən xüsusi bir

[7] Bəzən Perseptron adı tək bir TLU-dan ibarət kiçik şəbəkəni ifadə etmək üçün istifadə olunur.

neyron növü – *sürüşmə neyronu* (bias neuron) ilə təmsil edilir. İki inputlu və üç outputlu bir Perseptron Şəkil 10-5 -də təsvir edilib. Bu Perseptron eyni vaxtda nümunələri üç müxtəlif ikili təsnifatlarda(binary classes) təsnif edə bilər ki, bu da onu çoxçıxışlı təsnifedici (multioutput classifier) edir.



Şəkil 10-5. İki input neyronu, bir sürüşmə(bias) neyronu və üç output neyronundan ibarət Perseptronun arxitekturası.

Xətti cəbr sayəsində, Tənlik 10-2 bir süni neyron təbəqəsinin outpularını bir dəfəyə bir neçə nümunə üçün səmərəli şəkildə hesablamağa imkan verir.

Tənlik 10-2. Tam qoşulmuş təbəqənin çıxışlarının hesablanması

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

Bu tənlikdə:

- **X** – input xüsusiyyətlərinin (feature) matrisidir. Onun hər nümunə üçün bir sətri, hər xüsusiyyət üçün isə bir sütunu var.
- **W** – sürüşmə (bias) neyronundan gələnlər istisna olmaqla bütün bağlantı çəkisindən (connection weight) ibarət olan çəkilər matrisidir. Onun təbəqəsində hər input neyronu üçün bir sətri, hər süni neyron üçün bir sütunu var.
- **b** – sürüşmə neyronu ilə süni neyronlar arasındakı bağlantı çəkisindən ibarət sürüşmə vektorudur. Hər süni neyron üçün bir sürüşmə dəyəri mövcuddur.
- **ϕ** – *aktivasiya funksiyasıdır*. Əgər süni neyronlar TLU-dursa, bu funksiya addım funksiyası olur (lakin irəlidə digər aktivasiya funksiyalarını da müzakirə edəcəyik).

Perseptron necə öyrədilir? Rozenblatt tərəfindən təklif edilən Perseptron təlim alqoritmi əsasən *Hebb qaydasından* ilhamlanıb və Rozenblatt tərəfindən təklif edilib. Donald Hebb 1949-cu ildə yazdığı Davranışın təşkili (*The Organization of Behavior*) (Wiley) kitabında təklif edirdi ki, əgər bir bioloji neyron tez-tez digər bioloji neyronu işə salırsa, onların arasındakı əlaqə güclənir. Daha sonra Zieqrid Lovel bu fikri yaddaqalan bir ifadə ilə ümumiləşdirdi: "Birlikdə aktivləşən hüceyrələr, birlikdə əlaqələnirlər"; yəni, iki neyron eyni vaxtda aktiv olduqda onların arasındakı əlaqə çəkisi artır. Bu prinsip sonradan Hebb qaydası və ya Hebbian öyrənmə kimi tanındı. Perseptronlar, şəbəkənin proqnoz verdiyi zaman etdiyi səhvi nəzərə alan bu qaydanın bir forması istifadə edilərək öyrədilir. Perseptron öyrənmə qaydası xətaları azaltmağa kömək edən əlaqələri gücləndirir. Daha dəqiq desək , Perseptrona hər dəfə bir təlim nümunəsi təqdim olunur, hər nümunə üçün proqnoz verilir, Əgər output neyronu səhv proqnoz veribsə, həmin neyrona düzgün proqnozu dəstəkləyəcək inputlardan gələn bağlantı çəkiləri (connection weights) gücləndirilir. Bu qayda Tənlik 10-3-də göstərilib.

Tənlik 10-3. Perseptron öyrənmə qaydası (çəki yeniləmə)

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta\left(y_j - \hat{y}_j\right)x_i$$

Burada:

- $w_{i,j}$ i^{th} *input neyronu ilə* j^{th} *output neyronu arasındakı bağlantı çəkisi,*
- x_i mövcud tə lim nümunəsinin i^{th} *input dəyəri,*
- $\hat{y}_j$ j^{th} output neyronunun mövcud nümunə üçün verdiyi nəticə,
- y_j j^{th} output neyronunun mövud təlim nümunəsinin hədəf nəticəsidir,
- η– öyrənmə dərəcəsidir (learning rate).

Hər bir output neyronunun qərar sərhədi xəttidir. Buna görə də, Perseptronlar mürəkkəb nümunələri öyrənə bilmir (logistik reqressiya təsnifediciləri kimi). Lakin, əgər təlim nümunələri xətti ayırıcıdırsa (linearly separable), Rozenblatt göstərdi ki, bu alqoritm mütləq həllə[8] yaxınlaşacaq. Bu nəticə *Perseptron yaxınlaşma teoremi* (Perceptron Convergence Theorem) adlanır.

[8] Qeyd etmək lazımdır ki, bu həll unikal deyil: əgər verilənlər nöqtələri xətli (linearly) şəkildə ayrılabilirsə, onları ayıran sonsuz sayda ayırıcı xətt mövcuddur.

Əlavə olaraq, Scikit-Learn kitabxanası *Perceptron sinfini* təqdim edir hansı ki, tək-TLU şəbəkəsini reallaşdırır. Onu məsələn, iris datası (Fəsil 4-də təqdim olunmuşdur) üzərində istifadə etmək olar.

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris()
X = iris.data[:, (2, 3)]  # petal length, petal width
y = (iris.target == 0).astype(np.int)  # Iris setosa?

per_clf = Perceptron()
per_clf.fit(X, y)

y_pred = per_clf.predict([[2, 0.5]])
```

Perseptron öyrənmə alqoritminin *Stoxastik Qradient Enmə (Stochastic Gradient Descent)* ilə güclü şəkildə oxşarlığı olduğunu artıq müşahidə etmiş ola bilərsiniz. Əslində, Scikit-Learn kitabxanasındakı Perseptron sinfi, müəyyən hiperparametrlərlə qurulmuş bir *SGDClassifier*-ə ekvivalentdir:

loss="perceptron", learning_rate="constant", eta0=1 (öyrənmə dərəcəsi),penalty=None (heç bir regulyasiya yoxdur).

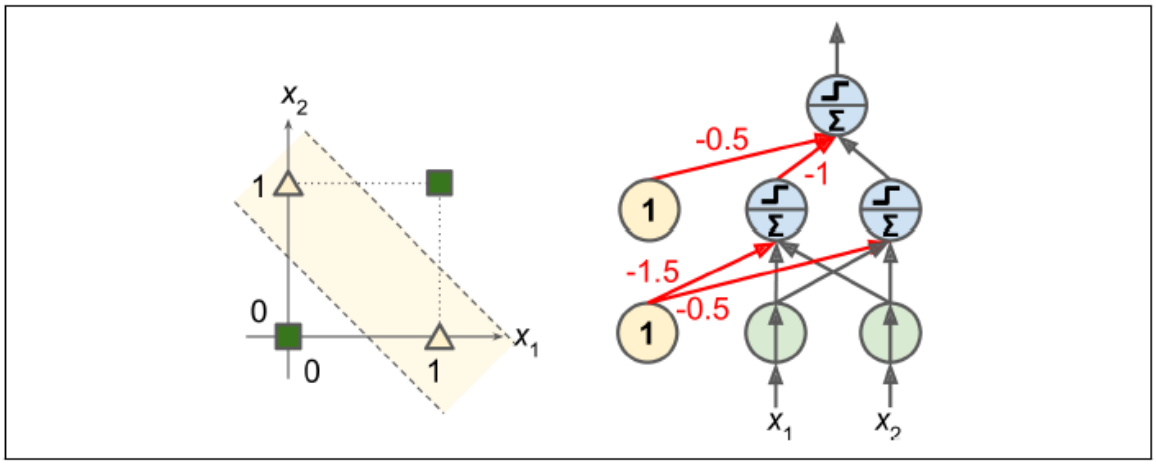
Qeyd etmək lazımdır ki, Logistik Reqressiya klassifikasiyasından fərqli olaraq, Perseptronlar sinif ehtimalı çıxarmırlar; onlar sadəcə sərt həddə (hard threshold) əsasında proqnoz verirlər. Bu isə Logistik Reqressiyanın, Perseptrona nisbətdə üstün tutulma səbəbidir.

1969-cu ildə Marvin Minski və Seymur Papert-in *Perceptrons* adlı monoqrafiyasında Perseptronların bir sıra ciddi zəif cəhətləri üzə çıxarıldı -xüsusilə, onların bəzi xırda məsələləri həll edə bilmədiyi vurğulandı.(Məsələn, *XOR (Exclusive OR)* təsnifat problemi Bax:şəkil 10-6-nın sol tərəfində göstərildiyi kimi). Bu eyni zamanda bütün digər xətti təsnifat modellərinə (məsələn, Logistik Reqressiya klassifikasiyasl) də aiddir. Lakin tədqiqatçılar Perseptronlardan daha çox şey gözlədiklərinə görə məyus oldular və onların bir hissəsi neyron şəbəkələrini tamamilə tərk edərək, məntiq, problem həlli və axtarış kimi daha yüksək səviyyəli məsələlərə yönəldilər.

Məlum oldu ki, Perseptronların bəzi məhdudiyyətlərini çoxqatlı Perseptonların yığılması vasitəsilə aradan qaldırmaq mümkündür. Yəni, bir neçə Perseptronu ardıcıl şəkildə yığmaqla əldə edilən ANN-ə *Çoxqatlı Perseptron (Multilayer Perceptron, MLP)* deyilir.

MLP artıq XOR problemini həll edə bilir. Bunu Şəkil 10-6-nın sağ tərəfində təsvir edilən MLP-nin outputunu hesablamaqla təsdiqləyə bilərsiniz: İnputlar (0, 0) və ya (1, 1) olduqda, şəbəkə

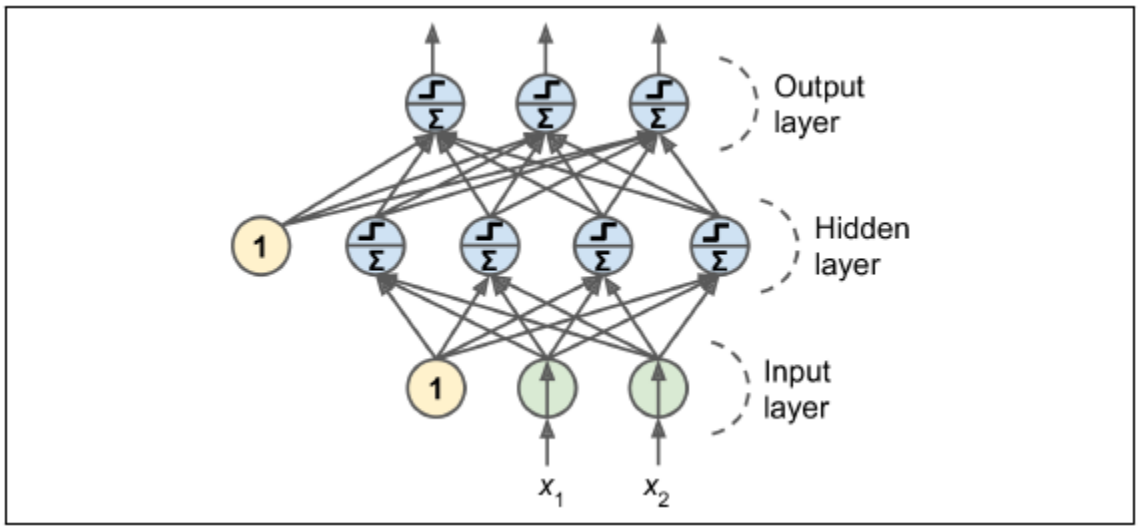
outputu 0 olur. İnputlar (0, 1) və ya (1, 0) olduqda, şəbəkə outputu 1 olur. Bütün əlaqələrin çəkisi 1-ə bərabərdir, yalnız şəkildə xüsusi olaraq göstərilən dörd əlaqənin çəkisi fərqlidir. Bu MLP-nin həqiqətən XOR problemini həll etdiyini yoxlamağa cəhd edin!



Şəkil 10-6. XOR klassifikasiya problemi və onu həll edən Çoxqatlı Perseptron (MLP)

Çoxqatlı Perseptron və Geri yayılma (Backpropogation)

MLP bir (ötürücü) *input qatından*, bir və ya bir neçə *gizli qatlar* adlanan TLU qatından və output qatı adlanan TLU-ların son qatından təşkil olunur (bax**:** Şəkil 10-7). İnput qatına yaxın olan qatlara adətən *aşağı qatlar*, outputlara yaxın olanlara isə *yuxarı qatlar* deyilir. Output qatı istisna olmaqla, hər bir qat sürüşmə(bias) neyronunu da ehtiva edir və tam şəkildə növbəti qatla əlaqələndirilir.



Şəkil 10-7. İki giriş, dörd neyronlu bir gizli qat və üç çıxış neyronundan ibarət Çoxqatlı Perseptronun arxitekturası (bu təsvirində sürüşmə neyronları göstərilmişdir, lakin adətən onlar dolayı (gizli) şəkildə qəbul olunur).

Siqnal yalnız bir istiqamətdə (inputlardan outputlara doğru) hərəkət edir, buna görə də bu arxitektura *önə doğru ötürücü neyron şəbəkəsi (FNN)* nümunəsidir.

Süni Neyron Şəbəkəsində(ANN) gizli qatların[9] dərin yığını olduqda, ona *dərin neyron şəbəkəsi (DNN-Deep Neural Network)* deyilir. Dərin Öyrənmə sahəsi DNN-ləri və daha ümumi halda dərin hesablama qatları ehtiva edən modelləri öyrənir. Buna baxmayaraq, bir çox insanlar neyron şəbəkələri (hətta dayaz olanları) nəzərdə tutduqda belə, Dərin Öyrənmə terminindən istifadə edirlər.

Uzun illər tədqiqatçılar Çoxqatlı Perseptronları (MLP) öyrətməyin yolunu axtarsalar da, uğurlu nəticə əldə edə bilmirdilər. Lakin 1986-cı ildə David Rumelhart, Cefriy Hinton və Ronald Uliams tərəfindən nəşr edilən mühüm bir məqalə[10] ilə *geri yayılma (backpropagation)* öyrətmə alqoritmi təqdim olundu və bu alqoritm bu gün də istifadə olunur.

Qısacası, bu, Q*radient Enmə (Gradient Descent*) üsuludur[11] (4-cü fəsildə təqdim olunmuşdur), hansı ki, qradiyentlərin avtomatik hesablanması üçün səmərəli texnikanın istifadəsi tətbiq edilir. Yalnız iki keçid vasitəsilə (bir irəli, bir geri) geri yayılma alqoritmi şəbəkənin xətasının bütün model parametrlərinə görə qradiyentini hesablaya bilir. Başqa sözlə, o müəyyən edir ki, xətanı azaltmaq üçün hər bir əlaqə çəkisi(connection weight) və hər bir sürüşmə termini (bias term) necə tənzimlənməlidir. Bu qradiyentlər əldə olunduqdan sonra sadəcə adi Qradient Enmə addımı icra edilir və proses şəbəkə həllə yaxınlaşana qədər təkrarlanır.

Qradiyentlərin avtomatik hesablanması *avtomatik differensiallaşdırma (automatic differentiation, autodiff)* adlanır. Müxtəlif üstünlüklərə və çatışmazlıqlara malik bir sıra autodiff texnikaları mövcuddur. Geri yayılma (backpropagation) alqoritmində istifadə olunan üsul isə *əks rejimli autodiff (reverse-mode autodiff*) adlanır. Bu üsul sürətli və dəqiqdir və xüsusilə çoxsaylı dəyişənlərə (məsələn, əlaqə çəkiləri) və az sayda çıxışa (məsələn, tək bir xəta funksiyasına) malik funksiyaların differensiallaşdırılması üçün çox

[9] 1990-cı illərdə iki və ya daha çox gizli qatı olan süni neyron şəbəkəsi (ANN) dərin hesab olunurdu. Hal-hazırda isə onlarla, hətta yüzlərlə qatdan ibarət ANN-lərlə tez-tez rastlaşmaq mümkündür, buna görə də "dərin" anlayışının dəqiq tərifi olduqca qeyri-müəyyəndir.

[10] David Rumelhart və başqaları, *"Learning Internal Representations by Error Propagation"* (Xətanın Yayılması ilə Daxili Təmsillərin Öyrənilməsi), Müdafiə Texniki İnformasiya Mərkəzi (Defense Technical Information Center) texniki hesabatı, sentyabr 1985.

[11] Bu texnika əslində müxtəlif sahələrdə fərqli tədqiqatçılar tərəfindən bir neçə dəfə müstəqil şəkildə ixtira olunmuşdur və onun başlanğıcı 1974-cü ildə Paul Verbos ilə qoyulmuşdur.

uyğundur. Əgər autodiff haqqında daha ətraflı öyrənmək istəsəniz, Əlavə D bölməsinə baxa bilərsiniz.

Gəlin bu alqoritmi bir qədər ətraflı nəzərdən keçirək:

- Alqoritm hər dəfə bir mini-dəst(mini-batch) (məsələn, hər biri 32 nümunə olan) üzərində işləyir və bütün təlim dəstindən dəfələrlə keçir. Hər belə keçidə *epoxa* deyilir.
- Hər mini-dəst(mini-batch) şəbəkənin giriş qatına ötürülür və o, məlumatı ilk gizli qata göndərir. Daha sonra alqoritm həmin qatda bütün neyronların çıxışlarını (mini-dəstdəki hər bir nümunə üçün) hesablayır. Nəticə növbəti qata ötürülür, orada çıxış (output) hesablanır və bu proses sonuncu qata – çıxış qatına qədər davam edir. Bu *mərhələyə irəli keçid (forward pass)* deyilir: proqnozvermə prosesinə tamamilə bənzəyir, sadəcə olaraq bütün aralıq nəticələr saxlanılır, çünki onlar geri keçid üçün lazımdır.
- Daha sonra alqoritm şəbəkənin çıxış xətasını ölçür (yəni, istənilən çıxışla şəbəkənin faktiki çıxışını müqayisə edən və xəta ölçüsünü qaytaran bir itki funksiyası(loss function) istifadə olunur).
- Sonra isə hər bir çıxış əlaqəsinin(output connection) xətaya nə dərəcədə təsir etdiyini hesablayır. Bu, analitik olaraq *zəncir qaydasının (chain rule)* tətbiqi ilə edilir (hesablama riyaziyyatının ən fundamental qaydalarından biridir). Bu üsul sayəsində mərhələ həm sürətli, həm də dəqiq olur.
- Alqoritm daha sonra hər bir aşağı qat bağlantısının səhv payına nə qədər təsir etdiyini ölçür, yenə zəncir qaydası (chain rule) tətbiq edərək, prosesi giriş qatına(input layer) qədər geriyə doğru davam etdirir. Əvvəl qeyd edildiyi kimi, bu geri keçid (reverse pass) şəbəkədəki bütün əlaqə çəkiləri üzrə səhv qradiyentini səmərəli şəkildə ölçür və səhv qradiyentinin şəbəkə boyunca geri ötürülməsini təmin edir (bu alqoritmin adının mənası da bundandır).
- Nəhayət, alqoritm hesablanan xətalı qradiyentlərdən istifadə edərək şəbəkədəki bütün əlaqə çəkilərini düzəltmək üçün Qradient Enmə (Gradient Descent) addımını icra edir.

Bu alqoritm o qədər əhəmiyyətlidir ki, onu yenidən ümumiləşdirib ifadə etmək faydalıdır: hər bir təlim nümunəsi üçün geri yayılma (backpropagation) alqoritmi əvvəlcə proqnoz verir (forward pass) və xətanı ölçür, sonra hər bir qatı geriyə doğru keçərək hər bir bağlantının xətadakı payını ölçür (reverse pass) və nəhayət, səhvi azaltmaq üçün bütün bağlantı çəkilərini tənzimləyir (Gradient Descent addımı).

Bütün gizli qatların əlaqə çəkilərini təsadüfi dəyərlərlə başlatmaq vacibdir, əks halda öyrədilmə prosesi uğursuz olacaq. Məsələn, əgər bütün çəki və sürüşmə (bias) dəyərlərini sıfırla başlatsanız, onda verilmiş qatda bütün neyronlar tamamilə eyni olacaq və geriyayılma (backpropagation) onları

tam eyni şəkildə dəyişdirəcək. Nəticədə onlar eyni qalacaq. Başqa sözlə, hər qatda yüzlərlə neyron olsa da, modeliniz sanki hər qatda yalnız bir neyrona sahibmiş kimi davranacaq: çox da ağıllı olmayacaq. Halbuki, çəkiləri təsadüfi şəkildə başlatdıqda *simmetriya pozulur* və geriyayılmaya imkan verir ki, fərqli növ neyronlardan ibarət komandanı öyrətsin.

Alqoritmin səmərəli şəkildə fəaliyyət göstərməsi üçün müəlliflər Çoxqatlı Perseptron (MLP) arxitekturasında mühüm dəyişiklik etmişlər: addım funksiyası (step function) logistik (siqmoid) funksiyası ilə əvəz olunmuşdur: $\sigma(z)=1/(1+exp(-z))$. Bu dəyişiklik əsaslı əhəmiyyət kəsb edirdi, çünki addım funksiyası yalnız sabit (düzxətli) seqmentlərdən ibarətdir və belə halda qradiyent mövcud olmur.(Qradient Enmə (Gradient Descent)düz səthlərdə irəliləyə bilmir) Halbuki, logistik funksiyanın hər yerdə davamlı və sıfırdan fərqli törəməsi mövcuddur və bu, Qradient Enməyə hər mərhələdə irəliləyiş imkanı yaradır. Qeyd etmək lazımdır ki, geri yayılma alqoritmi yalnız logistik funksiyada deyil, bir çox digər aktivasiya funksiyalarında da uğurla tətbiq edilə bilir. Onlardan ikisi xüsusilə geniş istifadə olunur:

Hiperbolik tangens funksiyası: tanh(z)= $2\sigma(2z)-1$

Logistik funksiya ilə oxşar olaraq, bu funksiya S formalı, davamlı və differensiallaa bilən funksiyadır. Bununla belə, hiperbolik tangensin output dəyər diapazonu $\{-1,1\}$ intervalını əhatə edir (logistik funksiyada bu diapazon [0,1] şəklindədir). Output dəyərlərinin bu aralıqda yerləşməsi, təlimin (training) başlanğıc mərhələsində hər bir qatın nəticələrinin təqribən sıfır ətrafında cəmlənməsinə səbəb olur ki, bu da çox vaxt prosesinin yaxınlaşma sürətini artırır.

Düzləşdirilmiş Xətti Vahid (Rectified Linear Unit, ReLU) funksiyası: $ReLU(z)=max(0,z)$

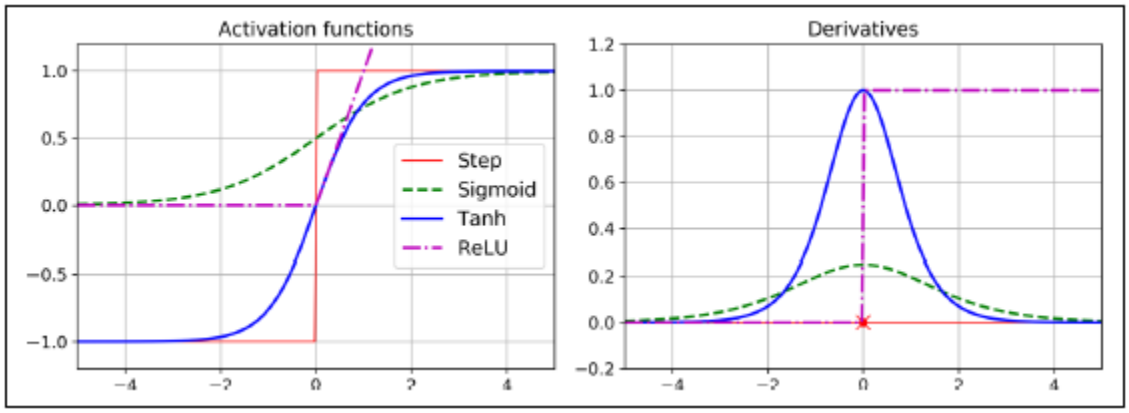
ReLU funksiyası davamlıdır, lakin təəssüf ki, $z=0$ nöqtəsində differensiallaşa bilmir, çünki bu nöqtədə meyl (slop) kəskin şəkildə dəyişir. Bu hal Qradient Enmə prosesində qeyri-sabitliyə (addımların "sıçramasına") səbəb ola bilər. Bundan əlavə, $z<0$ üçün törəmə sıfıra bərabərdir. Bununla belə, praktiki tətbiqlərdə ReLU çox səmərəli işləyir və hesablama baxımından sürətli olması səbəbindən neyron şəbəkələrdə təyin olunmuş (default) aktivasiya funksiyasına çevrilib[12]. Ən mühüm üstünlüklərindən biri isə odur ki, ReLU funksiyasının maksimal çıxış dəyəri mövcud deyil, bu da Qradient Enmə prosesində rast gəlinən bəzi problemlərin (məsələn, doyma effekti) azalmasına kömək edir (bu məsələyə 11-ci fəsildə geri dönüləcəkdir).

Bu məşhur aktivasiya funksiyaları və onların törəmələri Şəkil 10-8-də təsvir edilmişdir. Lakin burada sual yaranır: ümumiyyətlə aktivasiya funksiyalarına nə üçün ehtiyac duyulur?

Əgər bir neçə xətti çevrilməni ardıcıl şəkildə birləşdirsək, nəticədə yenə də yalnız bir xətti çevrilmə əldə edəcəyik. Məsələn, $f(x)=2x+3$ və $g(x)=5x-1$ olsun. Bu zaman bu iki xətti funksiyanı birləşdirdikdə yenə xətti funksiya alınır: $f(g(x))=2(5x-1)+3 = 10x+1$. Deməli, qatlar arasında qeyri-

[12] Bioloji neyronların təqribən siqmoid (S-şəkilli) aktivasiya funksiyasını reallaşdırdığı güman edilir, buna görə də tədqiqatçılar uzun müddət siqmoid funksiyalardan istifadə etmişlər. Lakin sonradan məlum olmuşdur ki, süni neyron şəbəkələrində (ANN-lərdə) ReLU funksiyası ümumiyyətlə daha yaxşı nəticə verir. Bu, bioloji analogiyanın yanıltıcı olduğu hallardan biri hesab edilir.

xətti (nonlinear) elementlər olmasa, istənilən qədər dərin qat yığsaq belə, şəbəkə faktiki olaraq birqatlı xətti modelə ekvivalent olacaq və mürəkkəb problemləri həll etmək mümkün olmayacaq. Əksinə, kifayət qədər böyük dərin neyron şəbəkəsi (DNN) qeyri-xətti aktivasiya funksiyaları ilə təchiz edildikdə, nəzəri cəhətdən istənilən davamlı funksiyaya yaxınlaşa bilir.



Şəkil 10-8. Aktivasiya funksiyaları və onların törəmələri

Artıq süni neyron şəbəkələrinin mənşəyi, onların arxitekturası və çıxışların necə hesablana bilməsi barədə biliklər əldə etmisiniz. Həmçinin, geri yayılma (backpropagation) alqoritmi ilə də tanış olmusunuz. Lakin bu mərhələdə əsas sual yaranır: süni neyron şəbəkələri ilə əslində nə etmək mümkündür?

Regressiya üçün Çoxqatlı Perseptronlar (MLP-lər)

İlk növbədə, Çoxqatlı Perseptronlar (MLP) reqressiya tapşırıqları üçün tətbiq oluna bilər. Əgər məqsəd yalnız bir dəyərin proqnozlaşdırılmasıdırsa (məsələn, evin müxtəlif xüsusiyyətləri əsasında onun qiymətinin təyin edilməsi), bu halda bir çıxış neyronu kifayət edir: həmin neyronun çıxışı proqnozlaşdırılan dəyər olacaq. Əgər söhbət çoxölçülü reqressiyadan gedirsə (yəni eyni anda bir neçə dəyərin proqnozlaşdırılması tələb olunursa), bu zaman hər bir çıxış ölçüsü üçün bir neyron nəzərdə tutulmalıdır. Məsələn, şəkildəki bir obyektin mərkəzini tapmaq üçün ikiölçülü koordinatların (x, y) proqnozlaşdırılması tələb olunur və buna görə də iki çıxış neyronu lazımdır. Əlavə olaraq, obyektin ətrafında çərçivə (bounding box) çəkmək üçün daha iki parametr – eni və hündürlüyü proqnozlaşdırmaq tələb olunur. Bu halda, ümumilikdə dörd çıxış neyronuna ehtiyac yaranır.

Ümumiyyətlə, reqressiya məqsədilə MLP qurarkən çıxış neyronları üçün aktivasiya funksiyasından istifadə etmək istəməsəniz belə ki, neyronların çıxışı istənilən dəyərləri almaqda azad olurlar. Lakin çıxışların mütləq müsbət olmasını təmin etmək lazım gəldikdə, çıxış qatında ReLU aktivasiya funksiyasını istifadə bilərsiniz. Alternativ olaraq, ReLU-nun sadə variantı olan *softplus* funksiyasından istifadə etmək mümkündür: softplus(z)=log(1+exp(z)).

O z mənfi olduqda 0-a yaxınlaşır, z müsbət olduqda isə z-yə yaxın olur. Əgər çıxışların(output) müəyyən bir intervalda yerləşməsini qaranti etmək istəyirsənsə, bu halda logistik və ya hiperbolik tangens funksiyaları tətbiq edilə bilər. Sonra isə etiketli(label) müvafiq olaraq uyğun diapazona miqyaslandırılmalıdır(scale): logistik funksiya üçün 0-dan 1-ə, hiperbolik tangens üçün isə 1-dən 1-ə.

Təlim (training) prosesində adətən istifadə olunan itki funksiyası (loss function) orta kvadratik xətadır (MSE- mean squared error). Lakin əgər təlim dəstində çoxlu kənar dəyərlər(outlier) mövcuddursa, bu halda orta mütləq xəta (MAE- mean absolute error) daha məqsədəuyğun ola bilər. Bundan başqa, hər iki yanaşmanın kombinasiyasını əks etdirən Huber itki funksiyasından da istifadə etmək olar.

Huber itki funksiyası səhvin dəyəri müəyyən bir δ həddən(threshold) (adətən 1 olur) kiçik olduqda kvadratik, həmin həddən böyük olduqda isə xətti xarakter daşıyır. Xətti hissə onu orta kvadratik xətaya(MSE) nisbətən kənar dəyərlərə (outlier) qarşı daha davamlı edir. Kvadratik hissə isə orta mütləq xətaya (MAE) nisbətən daha sürətli yaxınlaşma (konvergensiya) və daha yüksək dəqiqlik təmin edir.

Cədvəl 10-1 reqressiya üçün çoxqatlı perseptronunun (MLP) tipik arxitekturasını ümumiləşdirir.

Cədvəl 10-1. Reqressiya üçün tipik MLP arxitekturası

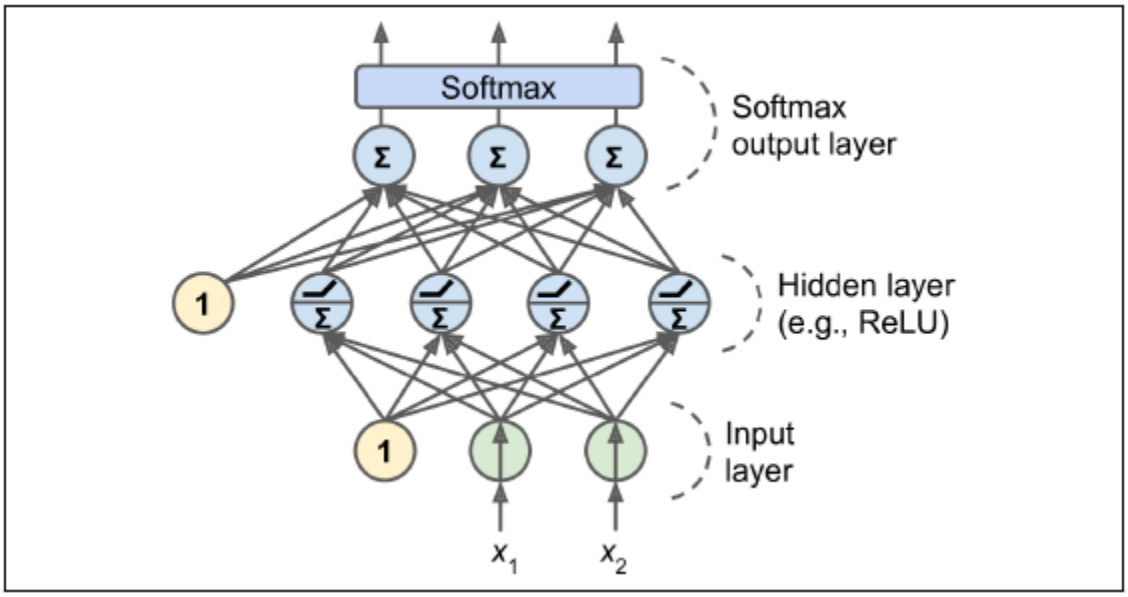
Hiperparametr	Tipik dəyər
Giriş neyronlarının sayı	Hər giriş xüsusiyyəti üçün bir neyron (məsələn, MNIST üçün 28 × 28 = 784)
Gizli qatların sayı	Problemdən asılı olaraq, adətən 1-dən 5-ə qədər
Hər gizli qatda neyron sayı	Problemdən asılı olaraq, adətən 10-dan 100-ə qədər
Çıxış(output) neyronlarının sayı	Hər proqnoz ölçüsü üçün bir neyron
Gizli aktivasiya funksiyası	ReLU (və ya SELU, bax: Fəsil 11)
Çıxış(output) aktivasiya funksiyası	Heç biri, və ya ReLU/softplus(yalnız müsbət çıxış üçün), logistik/tanh(məhdudlaşdırılmış çıxış üçün)
İtki funksiyası	MSE və ya MAE/Huber (əgər outlier varsa)

Klassifikasiya üçün Çoxqatlı Perseptronlar (MLP)

MLP-lər həmçinin klassifikasiya tapşırıqları üçün də tətbiq oluna bilər. İkili (binary) klassifikasiya məsələsində sadəcə olaraq logistik aktivasiya funksiyasından istifadə edən bir output neyronu kifayətdir: bu neyronun output-u 0 ilə 1 arasında bir dəyər alır və bu dəyər müsbət sinifin ehtimalının qiymətləndirilməsi kimi şərh olunur. Mənfi sinifin ehtimalı isə, bu ədədin 1-dən çıxılmasına bərabərdir.

MLP-lər həmçinin çoxetiketli (multilabel) ikili klassifikasiya tapşırıqlarını rahatlıqla idarə edə bilər (bax: Fəsil 3). Məsələn, sizdə elə bir elektron poçt təsnifat sistemi ola bilər ki, o, hər gələn e-poçtun spam və ya normal (ham) olduğunu, eyni zamanda isə onun təcili və ya qeyri-təcili olmasını eyni anda proqnozlaşdırsın. Bu halda, iki output neyronuna ehtiyacınız olacaq və onların hər ikisi logistik aktivasiya funksiyasından istifadə edəcək: birinci neyron e-poçtun spam olma ehtimalını,ikinci neyron isə onun təcili olma ehtimalını output kimi göstərəcək.Ümumi olaraq, hər bir müsbət sinif üçün ayrıca output neyronu təyin edilir. Qeyd etmək lazımdır ki, output ehtimalları mütləq olaraq 1-ə bərabər olmur, bu isə modelə etiketlərə istənilən kombinasiya üzrə proqnoz verməyə imkan yaradır: məsələn, təcili olmayan normal, təcili normal, təcili olmayan spam və hətta bəlkə də təcili spam.(Baxmayaraq ki, yüksək ehtimalla xəta ola bilər).

Əgər hər nümunə yalnız bir sinifə aid ola bilərsə və siniflərin sayı üç və ya daha çoxdursa (məsələn, rəqəm təsvirlərinin təsnifatı üçün 0-dan 9-a qədər siniflər), o zaman hər sinif üçün bir output neyronu nəzərdə tutulur. Bu halda bütün output qatında softmax aktivasiya funksiyası istifadə edilməlidir (bax: Şəkil 10-9). Softmax funksiyası(Bax: Fəsil 4) bütün ehtimalların 0 ilə 1 arasında olmasını və cəminin 1-ə bərabər olmasını təmin edir (siniflərin ekskluziv olduğu halda zəruridir). Bu yanaşma çoxsinifli (multiclass) klassifikasiya adlanır.



Şəkil 10-9. Klassifikasiya üçün müasir MLP (ReLU və softmax aktivasiya funksiyaları ilə)

İtki funksiyası ilə bağlı , ehtimal paylanmalarını proqnozlaşdırdığımız üçün Çarpaz entropiya(cross-entropy) itki funksiyası (digər adı ilə log loss, bax: Fəsil 4) ümumiyyətlə yaxşı seçimdir.

Cədvəl 10-2 klassifikasiya MLP-sinin tipik arxitekturasını ümumiləşdirir.

Cədvəl 10-2 Klassifikasiya üçün tipik MLP arxitekturası

Hiperparametr	İkili klasifikasiya	Çoxetiketli ikili klasifikasiya	Çoxsinifli klassifikasiya
İnput və gizli qatlar	Reqressiya ilə eyni	Reqressiya ilə eyni	Reqressiya ilə eyni
Output neyronlarının sayı	1	Hər etiket üçün 1	Hər sinif üçün 1
Output qatının aktivasiya funksiyası	Logistik	Logistik	Softmax
İtki funksiyası	Çarpaz entropiya	Çarpaz entropiya	Çarpaz entropiya

Davam etməzdən əvvəl sizə bu fəsilin sonunda yer alan 1-ci tapşırığa baxmağınızı tövsiyə edirəm. Siz müxtəlif neyron şəbəkə arxitekturaları ilə işləyəcək və onların nəticələrini *TensorFlow Playground* vasitəsilə vizuallaşdıracaqsınız. Bu, MLP-ləri (çoxqatlı perseptronları) və hiperparametrlərin (qatların və neyronların sayının, aktivasiya funksiyalarının və s.) təsirini daha yaxşı başa düşməyiniz üçün çox faydalı olacaq.

İndi Keras ilə MLP-ləri tətbiq etməyə başlamaq üçün lazım olan bütün anlayışlara sahibsiniz!

Keras ilə MLP-lərin tətbiqi

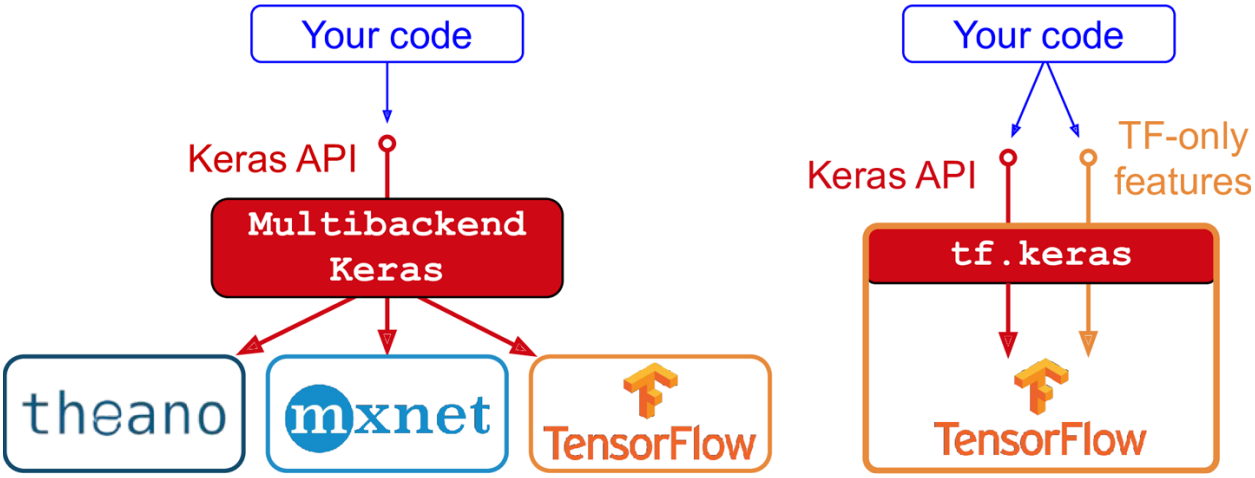
Keras — müxtəlif növ neyron şəbəkələrini asanlıqla qurmağa, öyrətməyə, qiymətləndirməyə və icra etməyə imkan verən yüksək səviyyəli Dərin Öyrənmə (Deep Learning) API-sidir. Onun sənədləşməsi (və ya spesifikasiyası) https://keras.io/ ünvanında mövcuddur.

Kerasın əsas (reference implementation) versiyası, "Keras" adlanır və Fransua Şolle tərəfindən bir tədqiqat layihəsinin [13] bir hissəsi kimi hazırlanmış, 2015-ci ilin mart ayında açıq mənbəli layihə kimi təqdim edilmişdir. Sadəliyi, çevikliyi və estetik dizaynı sayəsində Keras qısa müddətdə böyük populyarlıq qazanmışdır.

[13] Project ONEIROS — Açıq Sonlu Neyro-Elektron Ağıllı Robot Əməliyyat Sistemi).

Neyron şəbəkələrinin tələb etdiyi ağır hesablamaları yerinə yetirmək üçün bu əsas versiya hesablamalar üçün backend adlanan hesablama mühərrikinə əsaslanır. Hal-hazırda bu məqsədlə üç məşhur açıq mənbəli Dərin Öyrənmə kitabxanasından birini seçmək mümkündür: TensorFlow, Microsoft Cognitive Toolkit (CNTK), Theano. Buna görə də qarışıqlığın qarşısını almaq üçün bu əsas versiya *multibackend Keras* adlandırılır.

2016-cı ilin sonlarından etibarən başqa tətbiq versiyaları da təqdim edilmişdir. Artıq Kerası aşağıdakı mühitlərdə (environment) işlətmək mümkündür:Apache MXNet, Apple Core ML, JavaScript və TypeScript (veb-brauzerdə Keras kodunu işlətmək üçün), PlaidML (yalnız Nvidia deyil, müxtəlif GPU cihazlarında işləyə bilir). Bundan əlavə, TensorFlow artıq özünün daxilində Keras versiyasını tf.keras ilə birlikdə təqdim edir. Bu versiya yalnız TensorFlow-nu backend kimi dəstəkləyir, lakin bir sıra çox faydalı əlavə xüsusiyyətlərə malikdir (bax: Şəkil 10-10). Məsələn, TensorFlow Data API-sini dəstəkləyir ki, bu da məlumatların səmərəli yüklənməsi və əvvəlcədən emal edilməsini asanlaşdırır. Bu səbəbdən, bu kitabda biz f.keras-dan istifadə edəcəyik. Bununla belə, bu fəsildə TensorFlow-ya məxsus heç bir xüsusiyyətdən istifadə etməyəcəyik, ona görə də kodun digər Keras versiyalarında da (ən azı Python mühitində) kiçik dəyişikliklərlə, məsələn, import sətirlərini dəyişməklə problemsiz işləməsi mümkündür.



Şəkil 10-10. Keras API-nin iki tətbiqi: multibackend Keras (solda) və tf.keras (sağda)

Keras və TensorFlow-dan sonra ən məşhur Dərin Öyrənmə kitabxanası Facebook tərəfindən yaradılmış PyTorch kitabxanasıdır. Yaxşı xəbər odur ki, onun API-si Kerasınkına olduqca bənzəyir (bu da bir qədər Scikit-Learn və Chainer kitabxanalarından ilham alması ilə əlaqədardır). Buna görə də, Kerası öyrəndikdən sonra PyTorcha keçmək istəsəniz, bu sizə çətin olmayacaq.

PyTorch-un populyarlığı əsasən onun sadəliyi və əla sənədləşməsi sayəsində 2018-ci ildə sürətlə artmağa başladı,Halbuki TensorFlow 1.x-da bu xüsusiyyətlər o qədər güclü deyildi. Lakin TensorFlow 2 artıq PyTorch qədər sadə hesab edilir, çünki Kerası özünün rəsmi yüksək səviyyəli API-si kimi qəbul edib və onun tərtibatçıları digər API-ni də xeyli sadələşdirib və təmizləmişlər. Bu sənədləşmə də tamamilə yenidən təşkil edilmişdir və indi lazım olan məlumatı tapmaq çox daha asandır. Eyni şəkildə, PyTorch-un əsas zəif cəhətləri (məsələn, məhdud daşınma imkanı və

hesablama qrafikinin analizinin olmaması) PyTorch 1.0 versiyası ilə əsasən aradan qaldırılmışdır.Bu sağlam rəqabət hamı üçün faydalıdır.

İndi isə kod yazmaq vaxtıdır! tf.keras, TensorFlow ilə birlikdə təqdim olunduğuna görə, ilk addım olaraq "TensorFlow"nu quraşdıraq.

TensorFlow 2-nin quraşdırılması

Əgər siz Jupyter və Scikit-Learn kitabxanalarını 2-ci fəsildəki quraşdırma təlimatlarına uyğun şəkildə quraşdırmısınızsa, indi TensorFlow-u pip vasitəsilə quraşdıra bilərsiniz.

Əgər siz virtualenv istifadə edərək ayrıca (izolyasiya olunmuş) mühit yaratmısınızsa, əvvəlcə həmin mühiti aktivləşdirməyiniz lazımdır:

```
$ cd $ML_PATH                # Your ML working directory (e.g., $HOME/ml)
$ source my_env/bin/activate # on Linux or macOS
$ .\my_env\Scripts\activate  # on Windows
```

Daha sonra TensorFlow 2-ni quraşdırın (əgər virtualenv istifadə etmirsinizsə, bu halda administrator icazələrinə ehtiyacınız olacaq və ya əmrin sonuna --user parametrini əlavə etməlisiniz):

```
$ python3 -m pip install -U tensorflow
```

GPU dəstəyi üçün — bu kitabın yazıldığı dövrdə tensorflow əvəzinə tensorflow-gpu quraşdırmaq lazım idi. Lakin TensorFlow komandası həm tək CPU, həm də GPU ilə işləyən sistemləri dəstəkləyəcək vahid kitabxana üzərində işləyir.

GPU dəstəyini təmin etmək üçün hələ də əlavə kitabxanalar quraşdırmaq lazımdır (ətraflı məlumat üçün baxın: https://www.tensorflow.org/install).

GPU-lar haqqında daha ətraflı məlumatı 19-cu fəsildə nəzərdən keçirəcəyik.

Quraşdırmanı yoxlamaq üçün Python shell və ya Jupyter notebook açın, sonra TensorFlow və tf.keras kitabxanalarını import edin və onların versiyalarını print edin:

```
>>> import tensorflow as tf
>>> from tensorflow import keras
>>> tf.__version__
'2.0.0'
>>> keras.__version__
'2.2.4-tf'
```

İkinci versiya tf.keras tərəfindən tətbiq edilmiş Keras API versiyasıdır. Qeyd edək ki, onun adı '-tf' ilə bitir, bu, tf.keras-ın Keras API-sini tətbiq etməsilə yanaşı, TensorFlow-ya məxsus bəzi əlavə xüsusiyyətləri də ehtiva etdiyini göstərir.

İndi isə tf.keras-dan istifadə edək! Sadə bir şəkil təsnifləşdirici (image classifier) model qurmaqla başlayacayıq.

Silsilə API (Sequential API)-dən istifadə edərək Şəkil Təsnifləşdirici qurulması

Əvvəlcə bir dataset (məlumat toplusu) yükləməliyik. Bu fəsildə biz Fashion MNIST ilə işləyəcəyik hansı ki, bu dataset MNIST-in (3-cü fəsildə təqdim edilən) birbaşa əvəzedicisidir. Onun formatı MNIST ilə tam olaraq eynidir (70,000 rəngsiz (grayscale) şəkil var, hər biri 28 × 28 Piksel ölçüsündədir və 10 sinifə aiddir.). Lakin şəkillər burada əl ilə yazılmış rəqəmləri deyil, geyim əşyalarını təmsil edir. Buna görə də siniflər daha müxtəlifdir və tapşırıq MNIST-ə nisbətən xeyli daha çətindir.Məsələn, sadə bir xətti model (linear model) MNIST-də təxminən 92% dəqiqlik (accuracy) əldə edə bilir, lakin Fashion MNIST-də bu göstərici təxminən 83% olur.

Keras vasitəsilə dataseti yükləmək

Keras ümumi istifadə olunan bəzi datasetləri (məsələn, MNIST, Fashion MNIST və 2-ci fəsildə istifadə etdiyimiz California Housing) yükləmək üçün hazır yardımçı funksiyalar təqdim edir.

Gəlin Fashion MNIST datasetini yükləyək:

```
fashion_mnist = keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()
```

MNIST və ya Fashion MNIST datasetlərini Scikit-Learn əvəzinə Keras vasitəsilə yüklədikdə, mühüm fərq odur ki, hər bir şəkil 784 ölçülü birölçülü (1D) massiv kimi deyil, 28 × 28 ölçülü massiv kimi əks olunur. Bundan əlavə, piksel intensivlikləri 0.0-dan 255.0-a qədər onluqlarla (floats) deyil, 0-dan 255-ə qədər tam ədədlərlə (integers) ifadə edilir. Gəlin, təlim dəstinin (X_train_full) ölçüsünə (shape) və data tipinə baxaq:

```
>>> X_train_full.shape
(60000, 28, 28)
>>> X_train_full.dtype
dtype('uint8')
```

Qeyd edək ki, dataset artıq təlim (training) və test dəstlərinə bölünüb, amma validation dəsti yoxdur. Ona görə də, indi bir validation dəsti yaradacağıq. Əlavə olaraq, neyron şəbəkəni Qradient Enmə (Gradient Descent) ilə öyrədəcəyimiz üçün giriş xüsusiyyətlərini (input features) miqyaslandırmalıyıq(scale). Sadələşdirmək üçün, piksel intensivliklərini 0–1 aralığına salacağıq, bunun üçün onları 255.0-a bölməcəyik (bu həmçinin onları float tipinə çevirir):

```
X_valid, X_train = X_train_full[:5000] / 255.0, X_train_full[5000:] / 255.0
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
```

MNIST dataseti üçün, əgər etiketlər 5-ə bərabərdirsə, bu şəkilin əl ilə yazılmış 5 rəqəmini təmsil etdiyi anlamına gəlir. Sadədir. Lakin Fashion MNIST üçün hansı obyektlərlə işlədiyimizi bilmək üçün sinif adları siyahısına ehtiyacımız var:

```
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

Məsələn, təlim dəstindəki ilk şəkil bir paltonu təmsil edir:

```
>>> class_names[y_train[0]]
'Coat'
```

Şəkil 10-11 Fashion MNIST datasetindən bəzi nümunələri göstərir.

Şəkil 10-11. Fashion MNIST datasetindən nümunələr

Silsilə API (Sequential API) ilə modeli yaratmaq

İndi neyron şəbəkəni quraq! Aşağıda iki gizli qata malik bir klassifikasiya MLP-si göstərilib:

```
model = keras.models.Sequential()
model.add(keras.layers.Flatten(input_shape=[28, 28]))
model.add(keras.layers.Dense(300, activation="relu"))
model.add(keras.layers.Dense(100, activation="relu"))
model.add(keras.layers.Dense(10, activation="softmax"))
```

Gəlin bu kodu sətir-sətir nəzərdən keçirək:

• Birinci sətir bir Silsilə model yaradır. Bu, qatların ardıcıl şəkildə bir-birinə qoşulduğu ən sadə Keras modelidir. Buna Silsilə API (Sequential API) deyilir.

• Sonra birinci qatı qurub modelə əlavə edirik — bu Flatten qatıdır. Onun rolu hər bir input şəklini 1D massivə(array) çevirməkdir: əgər qat X girişini alırsa, o X.reshape(-1, 1) əməliyyatını yerinə yetirir. Bu qatın öyrənilən parametrləri yoxdur; o yalnız sadə ilkin emal (preprocessing) üçün

istifadə olunur. Modelin ilk qatı olduğu üçün, *input_shape* göstərilməlidir hansı ki, batch ölçüsü daxil deyil, yalnız nümunələrin forması daxildir. Alternativ olaraq birinci qat kimi *keras.layers.Input Layer* əlavə edib *input_shape=[28, 28]* təyin etmək olar.

• Daha sonra 300 neyronlu bir Sıx (Dense) gizli qat əlavə olunur. Bu qat ReLU aktivasiya funksiyasından istifadə edəcək. Hər Dense qatı öz çəkilər matrisi ilə idarə olunur — burada neyronlar ilə onların girişləri arasındakı bütün bağlantı çəkiləri saxlanılır. Həmçinin o bir sürüşmə vektorunu(vector of bias) idarə edir (hər neyron üçün). İnput məlumatı alanda qat uyğun tənliyi hesablayır (bax: Tənlik 10-2).

• Sonra 100 neyronlu ikinci bir Dense gizli qat əlavə edilir və həmçinin ReLU aktivasiyası istifadə olunur.

• Nəhayət, softmax aktivasiyası istifadə edilməklə 10 neyronlu (hər sinif üçün bir) Dense output qatı əlavə olunur (siniflər ekskluziv olduğu üçün).

Akivasiya="relu" yazmaq, *aktivasiya=keras.activations.relu* yazmaqla tam eyni mənaya gəlir. Kerasda aktivasiya funksiyalarının hamısı *keras.activations* paketində mövcuddur və bu kitabda onların bir çoxundan istifadə edəcəyik. Bütün aktivasiya funksiyalarının tam siyahısı ilə tanış olmaq üçün bu səhifəyə baxın: https://keras.io/activations/

Qatları bir-bir əlavə etmək əvəzinə, *Silsilə (Sequential)* modeli yaradarkən qatların siyahısını birbaşa ötürə bilərsiniz:

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, activation="relu"),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])
```

Keras.io saytındakı Kod Nümunələrindən İstifadə

keras.io saytında təqdim olunan kod nümunələri tf.keras ilə də problemsiz işləyəcək, lakin import əmrlərində bəzi dəyişikliklər etmək lazımdır. Məsələn, aşağıdakı keras.io kodunu nəzərdən keçirək:

```
from keras.layers import Dense
output_layer = Dense(10)
```

İmportları aşağıdakı kimi dəyişməlisiniz:

```
from tensorflow.keras.layers import Dense
output_layer = Dense(10)
```

Yaxud da tam yolları(full paths) istifadə edə bilərsiniz:

```
from tensorflow import keras
output_layer = keras.layers.Dense(10)
```

Bu yanaşma daha geniş görünə bilər, lakin bu kitabda bu üsuldan ona görə istifadə olunur ki, siz hansı paketlərin istifadə olunduğunu asanlıqla görə biləsiniz və standart siniflərlə (classes) xüsusi yaradılmış siniflər arasında qarışıqlığın qarşısı alınsın. Kod yazarkən isə əvvəlki, daha qısa yanaşmaya üstünlük verilir. Bir çox insanlar həmçinin aşağıdakı kimi yazılışdan istifadə edirlər:

from tensorflow.keras import layers hansı ki, ardınca *layers.Dense(10)* yazılır.

Modelin summary() metodu modeldəki bütün qatları göstərir[14]. Buraya hər bir qatın adı (qat yaradılarkən xüsusən qeyd olunmadığı təqdirdə avtomatik olaraq təyin olunur), output ölçüləri (output shape) (burada None dəyəri dəst(batch) ölçüsünün istənilən rəqəm ola biləcəyini bildirir)və parametr sayı daxildir. Summary modeldəki ümumi parametr sayı ilə bitir; bunlara həm öyrədilə bilən (trainable), həm də öyrədilə bilməyən (non-trainable) parametrlər daxildir.Bu nümunədə yalnız öyrədilə bilən parametrlər mövcuddur (öyrədilə bilməyən parametrlərə isə 11-ci fəsildə baxacağıq.)

```
>>> model.summary()
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 300)               235500
_________________________________________________________________
```

[14] Modelinizi şəkil formasında yaratmaq üçün keras.utils.plot_model() kodundan istifadə edə bilərsiniz.

```
dense_1 (Dense)              (None, 100)               30100
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1010
=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

Qeyd etmək lazımdır ki, Dense (sıx) qatlar çox vaxt çoxlu sayda parametrə malik olur. Məsələn, birinci gizli qatda 784 × 300 əlaqə çəkisi və əlavə olaraq 300 sürüşmə termini(bias term) var ki, bu da 235,500 parametr deməkdir! Bu qədər parametr modelə təlim məlumatlarını yaxşı öyrənmək üçün böyük çeviklik qazandırır, lakin eyni zamanda modelin əzbərləmə(overfitting) riskini də artırır xüsusən də təlim məlumatı az olduqda. Bu məsələyə daha sonra qayıdacağıq.

Modelin qatlarının siyahısını asanlıqla əldə etmək, müəyyən bir qatı indeksinə görə çağırmaq və ya adla əldə etmək mümkündür:

```
>>> model.layers
[<tensorflow.python.keras.layers.core.Flatten at 0x1324
 <tensorflow.python.keras.layers.core.Dense at 0x132414
 <tensorflow.python.keras.layers.core.Dense at 0x1356ba
 <tensorflow.python.keras.layers.core.Dense at 0x13240d
>>> hidden1 = model.layers[1]
>>> hidden1.name
'dense'
>>> model.get_layer('dense') is hidden1
True
```

Bir qatın bütün parametrlərinə onun *get_weights()* və *set_weights()* metodları vasitəsilə daxil olmaq mümkündür.Dense qatı üçün bu parametrlərə həm əlaqə çəkiləri, həm də sürüşmə terminləri daxildir:

```
>>> weights, biases = hidden1.get_weights()
>>> weights
array([[ 0.02448617, -0.00877795, -0.02189048, ..., -0.02766046,
         0.03859074, -0.06889391],
       ...,
       [-0.06022581,  0.01577859, -0.02585464, ..., -0.00527829,
         0.00272203, -0.06793761]], dtype=float32)
>>> weights.shape
(784, 300)
>>> biases
array([0., 0., 0., 0., 0., 0., 0., 0., 0., ...,  0., 0., 0.], dtype=float32)
>>> biases.shape
(300,)
```

Qeyd edək ki, Dense qatı əlaqə çəkilərini təsadüfi şəkildə başladır (hansı ki, bu, əvvəlki bölmədə qeyd etdiyimiz kimi simmetriyanı pozmaq üçün vacibdir) və sürüşmələr isə sıfır ilə başlanır ki, bu da tamamilə qəbuledilən dir. Əgər fərqli bir başlatma üsulu tətbiq etmək istəsəniz, qatı yaradarkən *kernel_initializer* (kernel – yəni əlaqə çəkilərinin matrisinin digər adıdır) və ya

bias_initializer parametrini təyin edə bilərsiniz.Başlatma üsullarını daha ətraflı şəkildə 11-ci fəsildə müzakirə edəcəyik. Tam siyahı ilə tanış olmaq üçün isə https://keras.io/api/layers/initializers/ səhifəsinə baxa bilərsiniz.

Çəkilər matrisinin(weight matrix) ölçüləri inputların sayından asılıdır. Bu səbəbdən, Silsilə modeldə ilk qat yaradılarkən *input_shape* parametrinin göstərilməsi tövsiyə olunur. Lakin əgər input ölçüsünü (input shape) əvvəlcədən göstərməsəniz, bu da qəbuledilə bilər: Keras modeli tam şəkildə qurmazdan əvvəl input ölçüsünü öyrənməyi sadəcə gözləyəcək. Bu ya modelə real data (məsələn, təlim zamanı) ötürülərkən, ya da *build()* metodu çağırıldıqda baş verəcək. Model tam şəkildə qurulana qədər (built) qatlar hələ çəkilərə sahib olmayacaq, bu səbəbdən bəzi əməliyyatları (məsələn, modelin nəticəsini print etmək və ya modeli saxlamaq) icra etmək mümkün olmayacaq. Beləliklə, əgər input ölçülərini əvvəlcədən bilirsinizsə, model yaradılarkən onu göstərmək ən yaxşısıdır.

Modelin tərtibi

Model yaradıldıqdan sonra, itki funksiyası və optimizator istifadəsi üçün *compile()* metodunu çağırmaq lazımdır. Əlavə olaraq, təlim və qiymətləndirmə mərhələlərində hesablanacaq əlavə metrikləri də siyahı şəklində göstərmək mümkündür:

```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

loss="sparse_categorical_crossentropy" istifadə etmək *loss=keras.losses.sparse_categorical_crossentropy* istifadə etməyə bərabərdir. Eyni şəkildə *optimizer="sgdə" optimizer=keras.optimizers.SGD()* ilə eynidir və *metrics=["accuracy"] isə metrics=[keras.metrics.sparse_categorical_accuracy]* ilə eyni nəticəni verir(bu itki funksiyasını istifadə etdikdə).Bu kitabda bir çox fərqli itki funksiyaları optimizatorlar və metriklər ilə tanış olacağıq. Tam siyahını https://keras.io/losses , https://keras.io/optimizers və https://keras.io/metrics ünvanlarından əldə edə bilərsiniz.

Bu kod üçün izah lazımdır. Əvvəlcə, *"sparse_categorical_crossentropy"* itkisini istifadə edirik, çünki etiketlərimiz aralıdır. (yəni, hər nümunə üçün yalnız hədəf sinif indeksi mövcuddur, bu halda 0-dan 9-a qədər) və siniflər ekskluzivdir. Əgər əvəzinə hər nümunə üçün hər sinifə aid bir hədəf ehtimalı olsaydı (məsələn, one-hot vektorlar, yəni sinif 3-ü ifadə etmək üçün [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.]), o zaman *"categorical_crossentropy"* itkisini istifadə etməli olardıq. Əgər biz ikili klassifikasiya həyata keçirirdiksə (bir və ya bir neçə ikili etiketlə), o zaman output qatında "softmax" aktivasiya funksiyası əvəzinə "sigmoid" (yəni logistik) aktivasiya funksiyasını istifadə etməli və itkini *"binary_crossentropy"* olaraq təyin etməliydik.

Əgər aralı etiketləri (yəni sinif indekslərini) one-hot vektor etiketlərinə çevirmək istəsəniz, *keras.utils.to_categorical()* funksiyasından istifadə edə bilərsiniz. Əks istiqamətdə çevirmək üçün isə *np.argmax()* funksiyasını *axis=1* funksiyası ilə istifadə etmək olar.

Optimizatorla əlaqəli "sgd" modelin sadə Stoxastik Qradient Enmə (Stochastic Gradient Descent) ilə öyrədiləcəyini bildirir. Başqa sözlə, Keras əvvəllər təsvir edilən geriyə yayılma(backpropagation) alqoritmini həyata keçirəcəkdir (yəni tərs rejimli avtomatik diferensiallaşma(autodiff) və Qradient Enmə). Daha səmərəli optimizatorlar haqqında isə 11-ci fəsildə danışacağıq (onlar Qradient Enmə hissəsini təkmilləşdirir, autodiff-i deyil).

SGD optimizatorundan istifadə edərkən öyrənmə dərəcəsini (learning rate) tənzimləmək vacibdir. Buna görə, adətən *optimizer="sgd"* əvəzinə *optimizer=keras.optimizers.SGD(lr=???)* istifadə edərək öyrənmə dərəcəsini təyin etmək istəyərsiniz, *çünki optimizer="sgd"* standart olaraq *lr=0.01* dəyərini götürür.

Nəhayət, bu bir təsnifedici olduğundan, onun öyrənmə(training) və qiymətləndirmə(evaluation) mərhələsində *"dəqiqlik" (accuracy)* ölçüsünü izləmək faydalıdır.

Modelin öyrədilməsi(training) və qiymətləndirilməsi(evaluating)

İndi model öyrədilməyə hazırdır. Bunun üçün sadəcə olaraq onun *fit()* metodunu çağırmaq kifayətdir:

```
>>> history = model.fit(X_train, y_train, epochs=30,
...                     validation_data=(X_valid, y_valid))
...
Train on 55000 samples, validate on 5000 samples
Epoch 1/30
55000/55000 [======] - 3s 49us/sample - loss: 0.7218     - accuracy: 0.7660
                                    - val_loss: 0.4973 - val_accuracy: 0.8366
Epoch 2/30
55000/55000 [======] - 2s 45us/sample - loss: 0.4840     - accuracy: 0.8327
                                    - val_loss: 0.4456 - val_accuracy: 0.8480
[...]
Epoch 30/30
55000/55000 [======] - 3s 53us/sample - loss: 0.2252     - accuracy: 0.9192
                                    - val_loss: 0.2999 - val_accuracy: 0.8926
```

Biz modelə giriş xüsusiyyətlərini(input feature) (X_train) və hədəf siniflərini(target classes) (y_train) ötürürük, həmçinin öyrənmə mərhələlərinin sayını (epoxa) təyin edirik (epoxa sayı təyin edilmədiyi halda 1 olacaq ki, bu da yaxşı bir nəticə üçün kifayət olmayacaq). Həmçinin, istəyə bağlı olaraq validasiya dəstini(mütləq deyil) də ötürə bilərik. Keras hər epoxanın sonunda bu dəstdə itkini və əlavə metrikləri ölçür ki, bu da modelin real performansını görmək üçün çox faydalıdır. Əgər öyrənmə dəstindəki performans validasiya dəstindəki performansdan xeyli yaxşıdırsa, bu, modelin yüksək ehtimalla öyrənmək üçün olan dəsti əzbərlədiyini (overfitting)

göstərir (və ya öyrənmə və validasiya dəstləri arasında məlumat uyğunsuzluğu kimi bir səhv mövcuddur).

Və bu qədər! Neyron şəbəkə öyrədildi[15]. Hər epoxa zamanı Keras indiyə qədər işlənmiş nümunələrin sayını (proqress bar ilə birlikdə), hər nümunəyə düşən orta öyrənmə vaxtını və həm öyrənmə, həm də validasiya dəstində itkini və dəqiqliyi (və ya tələb etdiyiniz hər hansı digər əlavə metrikləri) göstərir. Öyrənmə itkisinin azalmasını hansı ki, bu yaxşı bir əlamətdir, və ya 30 epoxadan sonra validasiya dəstində dəqiqliyin(validation accuracy) 89.26% səviyyəsinə çatdığını görə bilərsiz. Bu, öyrənmə dəstindəki dəqiqlikdən (training accuracy) çox uzaqda deyil, yəni overfitting çox görünmür.

validation_data arqumentindən istifadə edərək validasiya dəstini ötürmək əvəzinə, *validation_split* parametrini öyrənmə dəstinin Keras tərəfindən validasiya üçün istifadə olunmasını istədiyiniz nisbəti göstərmək üçün təyin edə bilərsiniz. Məsələn, *validation_split=0.1* Kerasa məlumatların son 10%-ni (qarışdırmazdan əvvəl) validasiya üçün istifadə etməsini bildirir.

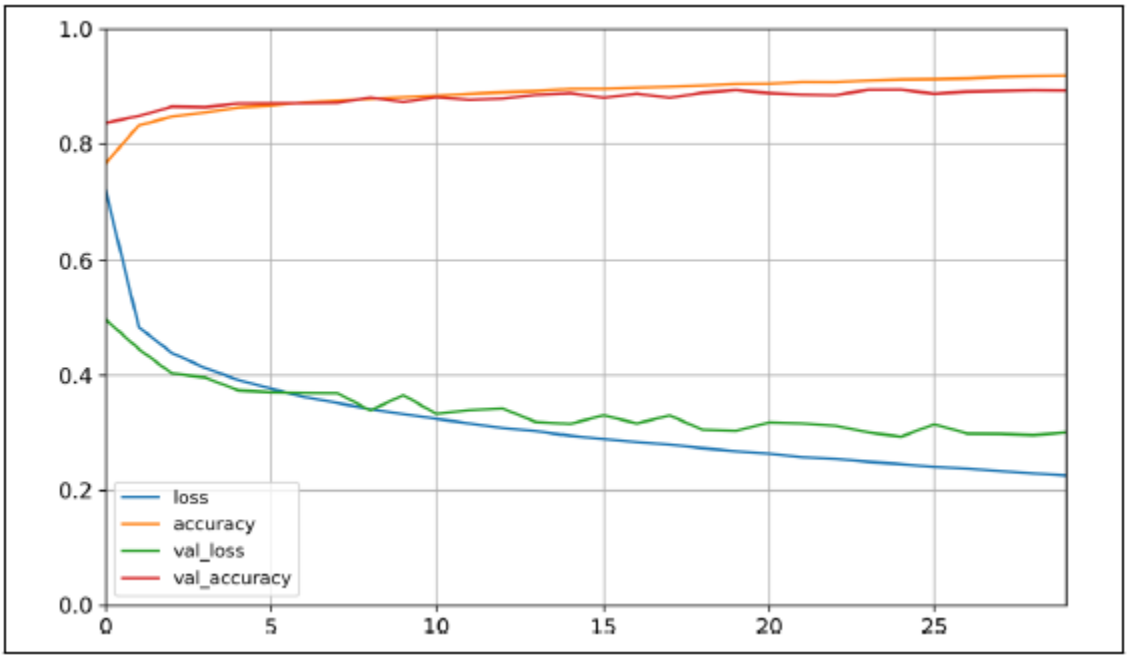
Əgər öyrənmə dəsti(training set) çox əyrilmişsə(skewed), bəzi siniflər həddindən artıq təmsil olunmuş, digərləri isə az təmsil olunmuşdursa, *fit()* metodunu çağırarkən *class_weight* arqumentini təyin etmək faydalı olar. Bu, az təmsil olunmuş siniflərə daha böyük, həddindən artıq təmsil olunmuş siniflərə isə daha kiçik çəki verər. Bu çəkilər itkini hesablarkən Keras tərəfindən istifadə olunur. Əgər nümunə səviyyəsində (per-instance) çəkilərə ehtiyac varsa, *sample_weight* arqumentini təyin edə bilərsiniz (əgər həm *class_weight*, həm də *sample_weight* verilsə, Keras onları bir-birinə vurur). Nümunə səviyyəsində çəkilər bəzi nümunələr ekspertlər tərəfindən, digərləri isə kütləvi qaynaq platforması ilə etiketlənmişdirsə faydalı ola bilər; belə hallarda əvvəlkilərə daha çox çəki vermək istəyə bilərsiniz. Həmçinin, validasiya dəsti üçün nümunə çəkilərini (lakin sinif çəkilərini verə bilməzsiniz) *validation_data tuple*-nın üçüncü elementi kimi əlavə etməklə verə bilərsiniz.

fit() metodu, öyrənmə parametrlərini (history.params), keçilən epoxaların siyahısını (history.epoch) və ən əsası hər epoxanın sonunda öyrənmə dəstində(training set) və varsa validasiya dəstində(validation test) ölçülən itki və əlavə metrikləri ehtiva edən bir lüğət (dictionary)(history.history) olan History obyektini qaytarır. Əgər bu lüğətdən istifadə edərək bir pandas DataFrame yaratsanız və onun *plot()* metodunu çağırsanız, 10-12-ci şəkildə göstərilən öyrənmə əyrilərini əldə edirsiniz.

[15] Əgər öyrənmə və ya validasiya dataları gözlənilən ölçüyə uyğun gəlmirsə, istisna baş verəcək. Bu, bəlkə də ən çox qarşılaşılan səhvdir, ona görə də səhv mesajına yaxşı bələd olmaq vacibdir. Mesaj əslində kifayət qədər aydındır: məsələn, əgər bu modeli *düzləşdirilmiş* (flattened)şəkillərdən ibarət bir array (X_train.reshape(-1, 784)) ilə öyrətməyə çalışsanız, aşağıdakı istisna ilə qarşılaşacaqsınız:
"ValueError: Error when checking input: expected flatten_input to have 3 dimensions, but got array with shape (60000, 784)."

```
import pandas as pd
import matplotlib.pyplot as plt

pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1) # set the vertical range to [0-1]
plt.show()
```



Şəkil 10-12. Öyrənmə əyriləri: hər epoxa üzrə ölçülən orta öyrənmə itkisi və dəqiqlik, həmçinin hər epoxanın sonunda ölçülən orta validasiya itkisi və dəqiqlik.

Öyrənmə itkisi və validasiya itkisi azalarkən, həm öyrənmədəki dəqiqliyin, həm də validasiyadakı dəqiqliyin öyrənmə zamanı ardıcıl olaraq artdığını görmək olar. Yaxşı! Üstəlik, təsdiqləmə əyriləri öyrənmə əyrilərinə yaxındır, bu isə əzbərləmənin(overfitting) çox olmadığını göstərir. Bu xüsusi halda, model öyrənmənin(training) əvvəlində validasiya dəstində öyrənmə dəstindən daha yaxşı performans göstərmiş kimi görünür. Lakin belə deyil: validasiya səhvi(validation error) hər epoxanın sonunda hesablanır, öyrənmə səhvi(training error) isə hər epoxa ərzində orta dəyər kimi hesablanır. Buna görə, öyrənmə əyrisi yarım epoxa sola sürüşdürülməlidir. Belə etdikdə, öyrənmə və validasiya əyrilərinin öyrənmənin əvvəlində demək olar ki, mükəmməl şəkildə üst-üstə düşdüyünü görəcəksiniz.

Öyrənmə əyrəsini çəkərkən, onu yarım epoxa sola sürüşdürmək lazımdır.

Öyrənmə dəstindəki(training set) performans sonda validasiya dəstindəki(validation set) performansı üstələyir, bu da, adətən, model kifayət qədər uzun müddət öyrədildikdə baş verir. Validasiya itkisi hələ də azalmaqda olduğuna görə modelin hələ tam konvergensiyaya çatmadığını anlaya bilərsiniz; buna görə, yəqin ki, öyrənməni davam etdirməli olacaqsınız. Bu isə sadəcə olaraq *fit()* metodunu yenidən çağırmaq qədər sadədir, çünki Keras əvvəlki mərhələdən öyrənməni davam etdirir (beləliklə, təxminən 89%-ə yaxın validasiya dəqiqliyinə çata bilərsən).

Əgər modelinizin performansı sizi qane etmirsə, hiperparametrləri tənzimləməyə qayıtmalısınız. İlk növbədə yoxlanılmalı olan parametr öyrənmə dərəcəsidir. Əgər bu kömək etməzsə, başqa bir optimizator sınayın (və hər hansı hiperparametri dəyişdirdikdən sonra öyrənmə dərəcəsini mütləq yenidən tənzimləyin). Əgər performans hələ də qənaətbəxş deyilsə, modelin hiperparametrlərini – məsələn, qatların sayını, hər qatdakı neyronların sayını və hər gizli qat üçün istifadə olunan aktivasiya funksiyalarının növlərini tənzimləməyə çalışın. Həmçinin, digər hiperparametrləri, məsələn, dəst(batch) ölçüsünü də dəyişmək olar (bu, *fit()* metodunda *batch_size* arqumenti vasitəsilə təyin olunur və standart olaraq 32-dir). Bu fəsilin sonunda hiperparametr tənzimləməsinə yenidən qayıdacağıq. Modelinizin validasiya dəqiqliyindən razı qaldıqdan sonra, modeli istehsalata buraxmazdan əvvəl(deploy the model to production) modelin ümumiləşdirmə xətasını (generalization error) təxmin etmək üçün onu test dəstində (test set) qiymətləndirməlisiniz. Bunu *evaluate()* metodu vasitəsilə asanlıqla etmək mümkündür (bu metod, həmçinin, *batch_size* və *sample_weight* kimi bir neçə əlavə arqumenti də dəstəkləyir; ətraflı məlumat üçün sənədləşməyə baxın):

```
>>> model.evaluate(X_test, y_test)
10000/10000 [==========] - 0s 29us/sample - loss: 0.3340 - accuracy: 0.8851
[0.3339798209667206, 0.8851]
```

2-ci fəsildə gördüyümüz kimi, test dəstində validasiya dəstinə nisbətən bir qədər aşağı nəticə əldə etmək adi haldır. Bunun səbəbi, hiperparametrlərin test dəstində deyil, validasiya dəstində tənzimlənməsidir. (Lakin bu nümunədə biz heç bir hiperparametr tənzimləməsi aparmadığımız üçün aşağı dəqiqlik sadəcə şanssızlıqdır.) Unutmayın ki, hiperparametrləri test dəsti üzərində dəyişdirməyə qarşı durmaq vacibdir əks halda ümumiləşdirmə xətası qiymətləndirməniz həddindən artıq optimist olacaq.

Modelin proqnoz üçün istifadəsi

Daha sonra, yeni nümunələr üzərində proqnozlar aparmaq üçün modelin *predict()* metodundan istifadə edə bilərik. Bizim əlimizdə real yeni nümunələr olmadığı üçün, sadəcə test dəstinin ilk üç nümunəsini istifadə edəcəyik.

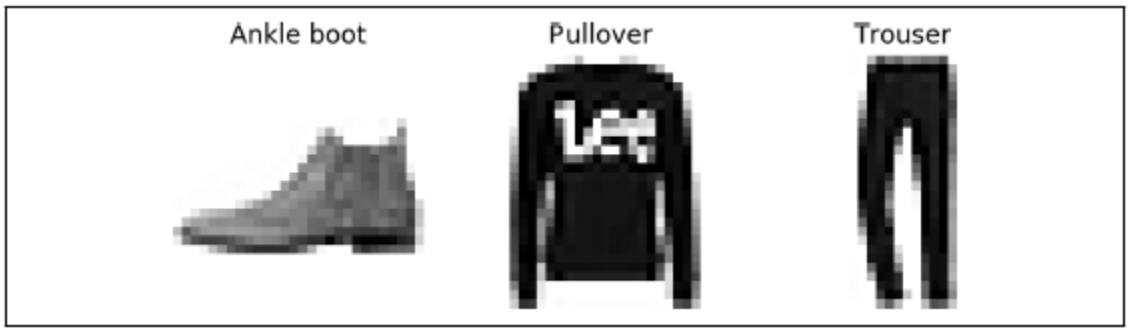
```
>>> X_new = X_test[:3]
>>> y_proba = model.predict(X_new)
>>> y_proba.round(2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.03, 0.  , 0.01, 0.  , 0.96],
       [0.  , 0.  , 0.98, 0.  , 0.02, 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ]],
      dtype=float32)
```

Göründüyü kimi, model hər bir nümunə üçün 0-dan 9-a qədər olan hər bir sinif üçün bir ehtimal dəyəri təxmin edir. Məsələn, birinci şəkil üçün model 9-cu sinifin (çəkmə) ehtimalını 96%, 5-ci sinifin (sandal) ehtimalını 3%, 7-ci sinifin (idman ayaqqabısı) ehtimalını isə 1% olaraq qiymətləndirir, digər siniflərin ehtimalları isə əhəmiyyətsiz dərəcədə azdır. Başqa sözlə, model "inanır" ki, birinci şəkil ayaqqabı növünə aiddir — çox güman ki, çəkmədir, lakin bəzən sandal və ya idman ayaqqabısı da ola bilər. Əgər yalnız ən yüksək təxmin edilmiş ehtimala malik sinfi (hətta bu ehtimal çox aşağı olsa belə) müəyyənləşdirmək istəyirsinizsə, bu halda *predict_classes()* metodundan istifadə edə bilərsiniz:

```
>>> y_pred = model.predict_classes(X_new)
>>> y_pred
array([9, 2, 1])
>>> np.array(class_names)[y_pred]
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')
```

Bu nümunədə təsnifləndirici əslində hər üç şəkli də düzgün klassifikasiya etmişdir (bu şəkillər 10-13-cü şəkildə göstərilmişdir).

```
>>> y_new = y_test[:3]
>>> y_new
array([9, 2, 1])
```



Şəkil 10-13. Düzgün klassifikasiya edilmiş Fashion MNIST datasından şəkillər.

İndi Silsilə API-dan istifadə edərək klassifikasiya MLP-sini qurmağı, öyrətməyi, qiymətləndirməyi və istifadə etməyi öyrəndiniz. Bəs reqressiya barədə nə demək olar?

Reqressiya MLP-sinin Silsilə API (Sequental API) ilə qurulması

Gəlin California housing məsələsinə keçək və onu reqressiya neyron şəbəkəsi ilə həll edək. Sadə üsulla datanı yükləmək üçün Scikit-Learn-in *fetch_california_housing*() funksiyasından istifadə edəcəyik. Bu dataset 2-ci fəsildə istifadə etdiyimizdən daha sadədir, çünki yalnız ədədi xüsusiyyətlərə malikdir (məsələn, ocean_proximity xüsusiyyəti yoxdur) və heç bir buraxılmış dəyər mövcud deyil. Datanı yüklədikdən sonra onu öyrənmə, validasiya və test dəstlərinə ayırırıq və bütün xüsusiyyətləri miqyaslaşdırırıq (scale).

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler


housing = fetch_california_housing()

X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_valid = scaler.transform(X_valid)
X_test = scaler.transform(X_test)
```

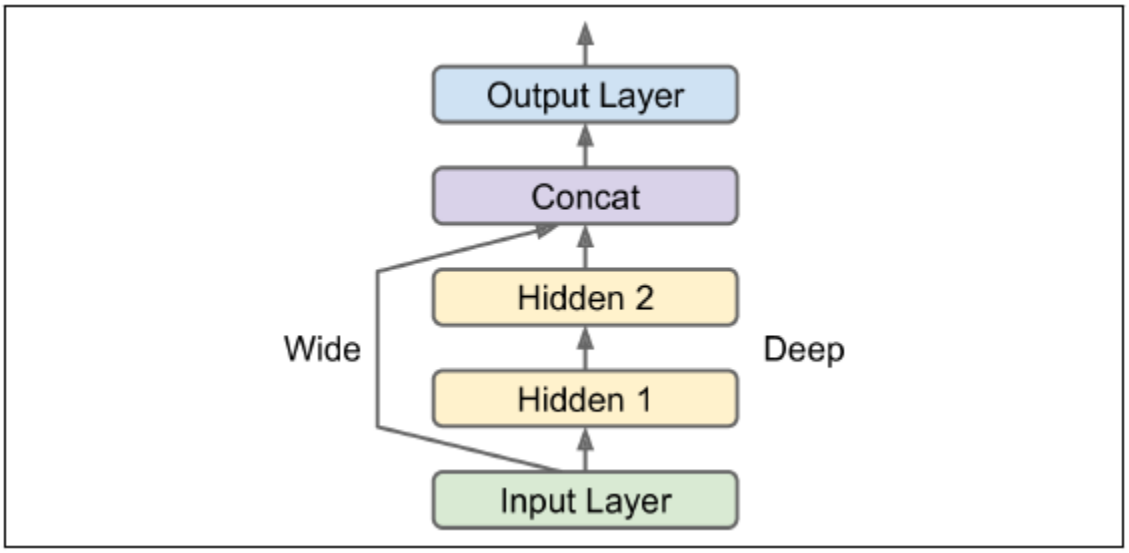
Silsilə API-dən istifadə edərək reqressiya MLP-sini qurmaq, öyrətmək, qiymətləndirmək və proqnozlar üçün istifadə etmək, klassifikasiya üçün etdiyimiz prosesə çox bənzəyir. Əsas fərqlər ondan ibarətdir ki, output qatı yalnız bir neyrona malikdir (çünki biz yalnız bir dəyəri proqnozlaşdırmaq istəyirik) və heç bir aktivasiya funksiyası istifadə etmir. İtki funksiyası isə orta kvadratik xətadır (*mean squared error*). Dataset kifayət qədər qarışıq (noisy) olduğuna görə, əzbərləmənin (overfitting) qarşısını almaq məqsədilə əvvəlkindən daha az neyronlu yalnız bir gizli qat (hidden layer) istifadə edirik.

```
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=X_train.shape[1:]),
    keras.layers.Dense(1)
])
model.compile(loss="mean_squared_error", optimizer="sgd")
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
mse_test = model.evaluate(X_test, y_test)
X_new = X_test[:3] # pretend these are new instances
y_pred = model.predict(X_new)
```

Gördüyünüz kimi, Silsilə API-dən istifadə etmək olduqca asandır. Lakin *Sequential* modelləri çox geniş yayılmış olsa da, bəzən daha mürəkkəb topologiyaya malik neyron şəbəkələri qurmaq və ya çoxsaylı input və outputlarla işləmək faydalı ola bilər. Bu məqsədlə Keras kitabxanası Funksional (API) Functional API adlı alternativ interfeys təqdim edir.

Funksional API (Functional API) ilə mürəkkəb modellərin qurulması

Qeyri-ardıcıl (nonsequential) neyron şəbəkələrə bir nümunə *"Wide & Deep" (geniş və dərin)* neyron şəbəkəsidir. Bu neyron şəbəkə arxitekturası ilk dəfə 2016-cı ildə Heng-Tze Çeng və digərləri tərəfindən təqdim edilmişdir[16]. Şəkil 10-14-də göstərildiyi kimi bu neyron şəbəkə arxitekturasında inputların hamısı və ya bir hissəsi birbaşa çıxış qatına (output layer) qoşulur. Belə arxitektura neyron şəbəkəyə eyni anda həm dərin qanunauyğunluqları (deep patterns)(dərin yollar-deep path istifadə etməklə, həm də sadə qaydaları (simple rules)(qısa yol-short path[17] vasitəsilə) öyrənməyə imkan verir. Əksinə, adi bir MLP (Multi-Layer Perceptron) bütün datanın bütün qatlar vasitəsilə keçməsinə məcbur edir; nəticədə, datadakı sadə nümunələr bu transformasiya silsiləsi zamanı təhrif oluna bilər.



Şəkil 10-14. Wide & Deep (Geniş və Dərin) neyron şəbəkəsi.

Gəlin belə bir neyron şəbəkəni quraraq California housing məsələsini həll edək:

```
input_ = keras.layers.Input(shape=X_train.shape[1:])
hidden1 = keras.layers.Dense(30, activation="relu")(input_)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.Concatenate()([input_, hidden2])
output = keras.layers.Dense(1)(concat)
model = keras.Model(inputs=[input_], outputs=[output])
```

Gəlin bu kodun hər sətrini nəzərdən keçirək:

[16] Heng-Tze Çeng və başqaları, *"Tövsiyə sistemləri üçün geniş və dərin öyrənmə"*, Dərin öyrənməyə əsaslanan tövsiyə sistemləri üzrə birinci seminarın materialları, 2016, s. 7–10.

[17] Qısa yol həmçinin neyron şəbəkəyə manual hazırlanmış xüsusiyyətlərin (features) verilməsi üçün də istifadə oluna bilər."

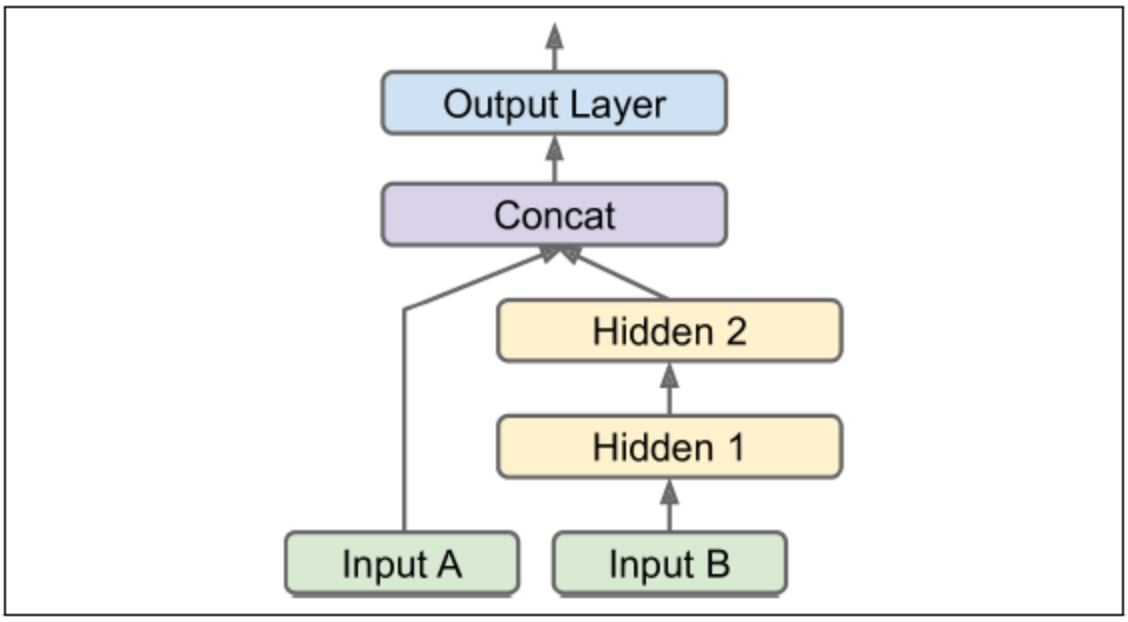
- İlk olaraq, *Input* obyektini[18] yaratmalıyıq. Bu, modelin alacağı inputun növünü, o cümlədən onun ölçüsünü (shape) və data tipini (dtype) müəyyən edir. Növbəti hissələrdə görəcəyimiz kimi, bir model əslində bir neçə inputa malik ola bilər.
- Sonra, 30 neyronlu *Dense qatı* yaradırıq və ReLU aktivasiya funksiyasından istifadə edirik. Qat yaradıldığı kimi, onu funksiyaya bənzər şəkildə çağırırıq və input obyektini ötürürük. Məhz buna görə də bu yanaşmaya Funksional API deyilir. Qeyd etmək lazımdır ki, burada Kerasa yalnız qatların necə birləşdiriləcəyini göstəririk; hələ heç bir real data işlənmir.
- Daha sonra ikinci gizli qatı (hidden layer) yaradırıq və yenidən onu funksiyaya bənzər şəkildə istifadə edirik. Burada ikinci qatın inputuna birinci gizli qatın outputu ötürülür.
- Sonra, Birləşdirmə (*Concatenate0* qatını yaradırıq və onu funksiyaya bənzər şəkildə çağıraraq giriş və ikinci gizli qatın outputunu birləşdiririk. İstəsəniz, *keras.layers.concatenate()* funksiyasından da istifadə edə bilərsiniz; bu funksiya həm *Concatenate* qatını yaradır, həm də onu verilmiş inputlarla dərhal çağırır.
- Daha sonra, yalnız bir neyrona malik və heç bir aktivasiya funksiyası istifadə etməyən çıxış qatını (output layer) yaradırıq; Output qatını funksiyaya bənzər şəkildə çağırırıq və ona birləşmənin nəticəsini ötürürük.
- Sonda, hansı input və outputların istifadə olunacağını göstərərək *Keras Model*i yaradırıq.

Keras modelini yaratdıqdan sonra proses əvvəlki kimi olur və belə ki, təkrarlamağa ehtiyac olmur: modeli tərtib etmək(compile), öyrətmək, qiymətləndirmək və proqnozlar üçün istifadə etmək lazımdır.

Bəs əgər xüsusiyyətlərin(features) bir hissəsini geniş yol (wide path), digər bir hissəsini isə dərin yol (deep path) üzrə göndərmək istəsək (mümkün qədər üst-üstə düşə bilər, bax Şəkil 10-15) Bu halda, bir həll yolu çoxsaylı inputlar istifadə etməkdir. Məsələn, geniş yol üzrə beş xüsusiyyət göndərmək istəsək (xüsusiyyətlər 0-dan 4-ə qədər) və dərin yol üzrə altı xüsusiyyət (xüsusiyyətlər 2-dən 7-yə qədər) göndərmək istəsək:

```
input_A = keras.layers.Input(shape=[5], name="wide_input")
input_B = keras.layers.Input(shape=[6], name="deep_input")
hidden1 = keras.layers.Dense(30, activation="relu")(input_B)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input_A, hidden2])
output = keras.layers.Dense(1, name="output")(concat)
model = keras.Model(inputs=[input_A, input_B], outputs=[output])
```

[18] *input_* adı Python-un daxili *input()* funksiyasını kölgələməmək üçün istifadə olunur.



Şəkil 10-15. Çoxsaylı girişlərin idarə edilməsi.

Kod aydındır. Xüsusilə model bir qədər mürəkkəb olduqda, ən azı ən vacib qatları adlandırmaq tövsiyə olunur. Qeyd edək ki, modeli yaradarkən *inputs=[input_A, input_B]* kimi təyin etdik. İndi modeli adi qaydada formalaşdıra bilərik, lakin *fit()* metodunu çağırdığımız zaman tək bir input matrisi (X_train) ötürmək əvəzinə, hər input üçün bir matris olmaqla iki matrisi (X_train_A, X_train_B) ötürməliyik[19]. Eyni qayda *evaluate()* və *predict()* metodlarını çağırarkən X_valid, X_test və X_new üçün də keçərlidir:

```
model.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=1e-3))

X_train_A, X_train_B = X_train[:, :5], X_train[:, 2:]
X_valid_A, X_valid_B = X_valid[:, :5], X_valid[:, 2:]
X_test_A, X_test_B = X_test[:, :5], X_test[:, 2:]
X_new_A, X_new_B = X_test_A[:3], X_test_B[:3]

history = model.fit((X_train_A, X_train_B), y_train, epochs=20,
                    validation_data=((X_valid_A, X_valid_B), y_valid))
mse_test = model.evaluate((X_test_A, X_test_B), y_test)
y_pred = model.predict((X_new_A, X_new_B))
```

Bir çox hallarda çoxsaylı outputa sahib olmaq istəyə bilərsiniz:

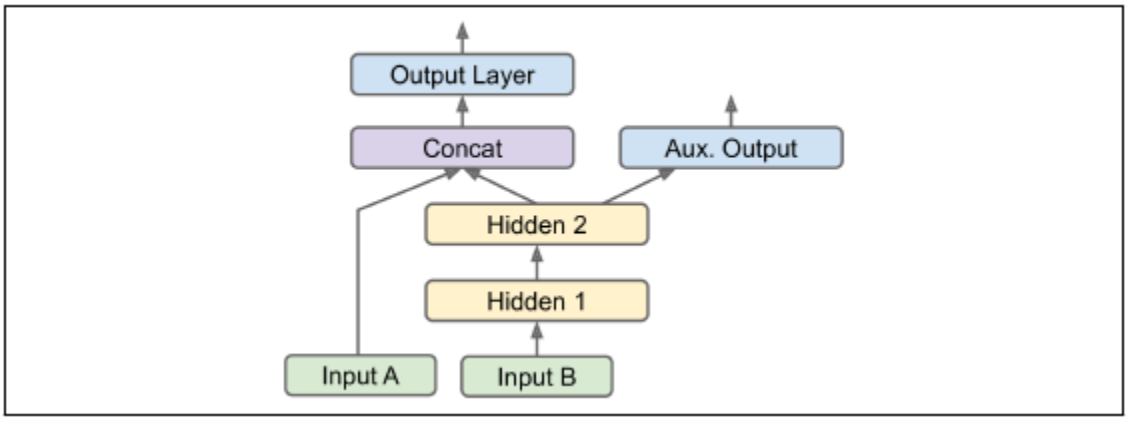
• Tapşırığın özü bunu tələb edə bilər. Məsələn, bir şəkildəki əsas obyektin yerini müəyyənləşdirmək və onu təsnifləşdirmək istəyə bilərsiniz. Bu, həm reqressiya tapşırığıdır

[19] Alternativ olaraq, input adlarını input dəyərləri ilə lüğət(dictionary) şəklində ötürə bilərsiniz, məsələn: {"wide_input": X_train_A, "deep_input": X_train_B}.Bu üsul xüsusilə çoxsaylı inputlar olduqda faydalıdır, çünki inputların sırasını qarışdırmağın qarşısını alır.

(obyektin mərkəzinin koordinatlarını, həmçinin onun enini və hündürlüyünü tapmaq), həm də klassifikasiya tapşırığıdır.

• Eyni data əsasında bir neçə müstəqil tapşırığınız ola bilər. Əlbəttə, hər tapşırıq üçün ayrıca bir neyron şəbəkə öyrədə(train) bilərsiniz, lakin bir çox hallarda hər tapşırıq üçün ayrıca outputa malik tək bir neyron şəbəkəni öyrətməklə bütün tapşırıqlarda daha yaxşı nəticə əldə etmək mümkündür. Bunun səbəbi, neyron şəbəkənin müxtəlif tapşırıqlarda faydalı olacaq ümumi xüsusiyyətləri datadan öyrənə bilməsidir. Məsələn, üz şəkilləri üzərində çoxtapşırıqlı təsnifat aparmaq olar: bir output şəxsin üz ifadəsini (gülümsəyir, təəccüblənib və s.) müəyyənləşdirə bilər, digər output isə şəxsin eynək taxıb-taxmadığını aşkar edə bilər.

• Digər bir istifadə halı isə bunu nizamlama (regularization) texnikası kimi tətbiq etməkdir (yəni modelin həddindən artıq öyrənməsini (overfitting) azaltmaq və ümumiləşdirmə qabiliyyətini artırmaq məqsədi daşıyan bir təlim(training) məhdudiyyəti). Məsələn, neyron şəbəkə arxitekturasına bəzi köməkçi outputlar əlavə edə bilərsiniz (bax: Şəkil 10-16). Bu, şəbəkənin təməl hissəsinin öz-özlüyündə faydalı bir şey öyrənməsini təmin edir, yəni şəbəkənin qalan hissəsindən tam asılı olmadan öyrənməni təşviq edir.



Şəkil 10-16. Bir neçə outputun idarə olunması, bu nümunədə nizamlama(regularization) məqsədilə köməkçi outputun əlavə olunması göstərilir.

Əlavə outputlar əlavə etmək olduqca asandır: sadəcə onları uyğun qatlara birləşdirin və modelinizin output siyahısına əlavə edin. Məsələn, aşağıdakı kod Şəkil 10-16-da göstərilən şəbəkəni qurur:

```
[...] # Same as above, up to the main output layer
output = keras.layers.Dense(1, name="main_output")(concat)
aux_output = keras.layers.Dense(1, name="aux_output")(hidden2)
model = keras.Model(inputs=[input_A, input_B], outputs=[output, aux_output])
```

Hər bir outputun öz itki funksiyası olmalıdır. Buna görə də modeli formalaşdırarkən, itkilərin siyahısını verməliyik[20] (Əgər yalnız bir itki funksiyası verilsə, Keras onun bütün outputlara tətbiq olunacağını qəbul edəcək.). Standart olaraq, Keras bütün bu itkiləri hesablayır və yekun itki kimi istifadə etmək üçün onları cəmləyir.Lakin biz əsas outputa köməkçi outputdan daha çox önəm veririk çünki köməkçi output sadəcə nizamlama (regularization) məqsədilə əlavə edilib. Buna görə, əsas outputun itkisinə daha böyük çəki vermək istəyirik. Yaxşı ki, Kerasda modeli formalaşdırarkən bütün itki çəkilərini təyin etmək mümkündür:

```
model.compile(loss=["mse", "mse"], loss_weights=[0.9, 0.1], optimizer="sgd")
```

İndi modeli öyrədərkən, hər bir output üçün etiketlər(labels) təqdim etməliyik. Bu nümunədə həm əsas output, həm də köməkçi output eyni şeyi proqnozlaşdırmağa çalışır, buna görə onlar eyni etiketlərdən istifadə etməlidirlər. Beləliklə, y_train əvəzinə (y_train, y_train) ötürməliyik (Eyni qayda y_valid və y_test üçün də keçərlidir).

```
history = model.fit(
    [X_train_A, X_train_B], [y_train, y_train], epochs=20,
    validation_data=([X_valid_A, X_valid_B], [y_valid, y_valid]))
```

Modeli qiymətləndirdikdə(evaluate), Keras ümumi itkini (total loss) və həmçinin hər bir output üçün ayrıca itkiləri (individual losses) geri qaytaracaq.

```
total_loss, main_loss, aux_loss = model.evaluate(
    [X_test_A, X_test_B], [y_test, y_test])
```

Eyni şəkildə, predict() metodu da hər bir output üçün proqnozları ayrıca şəkildə qaytaracaq.

```
y_pred_main, y_pred_aux = model.predict([X_new_A, X_new_B])
```

Gördüyünüz kimi, Functional API vasitəsilə istədiyiniz istənilən tipdə arxitekturanı asanlıqla qurmaq mümkündür. İndi isə, Keras modellərini qurmağın son bir üsuluna baxaq.

Dinamik modellərin yaradılması üçün Subklass API (Subclass API) -dən istifadə

Həm Silsilə API, həm də Funksional API bəyan edicidir: yəni əvvəlcə istifadə ediləcək qatları və onların necə birləşdiriləcəyini deklarasiya edirsiniz, sonra isə modeli təlim və ya nəticə üçün data ilə təmin edə bilərsiniz.Bu yanaşmanın bir çox üstünlükləri var: Modeli asanlıqla yadda saxlamaq, klonlamaq və paylaşmaq mümkündür; Onun strukturunu vizuallaşdırmaq və analiz etmək asandır; Çərçivə(Framework) avtomatik olaraq formaları və tipləri müəyyən edə bilir, bu da səhvləri erkən (yəni modelə real məlumat daxil olmadan) aşkarlamağa imkan verir; Həmçinin, bütün model statik

[20] Alternativ olaraq, hər output adını uyğun itki funksiyası ilə lüğət (dictionary) şəklində ötürə bilərsiniz. İnputlarda olduğu kimi, bu üsul çoxsaylı outputlar olduqda da faydalıdır, çünki sıra səhvlərinin qarşısını alır. Həmçinin, itki çəkiləri və metriklər(biraz müzakirə olunub) də lüğət vasitəsilə təyin oluna bilər.

qatlar qrafiki olduğuna görə debug prosesi də nisbətən sadədir. Lakin bu yanaşmanın mənfi tərəfi məhz budur: o statikdir.

Bəzi modellər iterasiyalar (loops), dəyişən ölçülər (varying shapes), şərti budaqlanmalar (conditional branching) və digər dinamik davranışlar tələb edir. Belə hallarda və ya sadəcə olaraq daha imperativ proqramlaşdırma üslubuna üstünlük verirsinizsə Subklass API istifadə olunur.

Sadəcə Model sinfindən törəyən yeni bir sinif yaradın. Ehtiyac duyduğunuz qatları konstrukturda yaradın və onları istədiyiniz hesablamaları yerinə yetirmək üçün *call()* metodunda istifadə edin. Məsələn, aşağıdakı WideAndDeepModel sinfindən bir nümunə yaratmaq, az öncə Funksional API ilə qurduğumuz modelə ekvivalent bir model verir. Sonra onu bizim etdiyimiz kimi tərtib etmək, qiymətləndirmək və proqnozlaşdırmaq mümkündür:

```
class WideAndDeepModel(keras.Model):
    def __init__(self, units=30, activation="relu", **kwargs):
        super().__init__(**kwargs) # handles standard args (e.g., name)
        self.hidden1 = keras.layers.Dense(units, activation=activation)
        self.hidden2 = keras.layers.Dense(units, activation=activation)
        self.main_output = keras.layers.Dense(1)
        self.aux_output = keras.layers.Dense(1)

    def call(self, inputs):
        input_A, input_B = inputs
        hidden1 = self.hidden1(input_B)
        hidden2 = self.hidden2(hidden1)
        concat = keras.layers.concatenate([input_A, hidden2])
        main_output = self.main_output(concat)
        aux_output = self.aux_output(hidden2)
        return main_output, aux_output

model = WideAndDeepModel()
```

Bu nümunə Funksional API-yə çox bənzəyir, lakin burada girişləri yaratmağa ehtiyac yoxdur, sadəcə onları *call()* metodunun input arqumenti vasitəsilə istifadə edirik. və konstruktorda qatların yaradılması onların *call()* metodu daxilində istifadəsindən ayrıca həyata keçirilir.[21] Əsas fərq isə ondadır ki, *call()* metodunda istədiyiniz demək olar ki, hər şeyi edə bilərsiniz *for* iterasiyaları, if şərtləri, aşağı səviyyəli TensorFlow əməliyyatları və s.Yəni, yaradıcılığınızın sərhədi yoxdur(bax: 12-ci fəsil). Bu xüsusiyyət onu yeni ideyalar üzərində eksperiment aparan tədqiqatçılar üçün ideal bir API edir.

Lakin bu əlavə çeviklik müəyyən mənfi cəhətlərlə gəlir: Modelin arxitekturası *call()* metodunun içində gizlənmiş olur, buna görə Keras onu asanlıqla analiz edə bilmir; Modeli asanlıqla saxlamaq və ya klonlamaq mümkün olmur; *summary()* metodunu çağırdıqda yalnız qatların siyahısı göstərilir, lakin onların bir-biri ilə necə əlaqələndirildiyi barədə məlumat verilmir;Həmçinin, Keras

[21] Keras modellərinin output adlı bir atributu olduğundan, əsas output qatı üçün bu adı istifadə etmək olmaz; buna görə də onu main_output olaraq dəyişdirdik.

əvvəlcədən tipləri və formaları yoxlaya bilmir, bu da səhv ehtimalını artırır.Ona görə də, əgər sizə bu əlavə çeviklik mütləq lazım deyilsə, Silsilə API və ya Funksional API ilə işləmək daha məqsədəuyğundur.

Keras modelləri adi qatlar kimi istifadə oluna bilər, buna görə də onları asanlıqla birləşdirərək mürəkkəb arxitekturalar qurmaq mümkündür.

Artıq Keras vasitəsilə neyron şəbəkələri necə qurmaq və öyrətmək (train) lazım olduğunu bildiyinizə görə, indi onları saxlamağı öyrənmək istəyəcəksiniz!

Modelin saxlanılması və bərpası

Silsilə API və ya Funksional API istifadə etdikdə, öyrədilmiş (train) Keras modelini saxlamaq çox sadədir:

```
model = keras.models.Sequential([...]) # or keras.Model([...])
model.compile([...])
model.fit([...])
model.save("my_keras_model.h5")
```

Keras HDF5 formatından istifadə edərək modelin həm arxitekturasını (yəni bütün qatların hiperparametrlərini), həm də hər bir qat üçün bütün parametrlərin dəyərlərini (məsələn, əlaqə çəkiləri(connection weights) və sürüşmələr(bias)) saxlayır. Bundan əlavə, optimizer də (onun hiperparametrləri və mövcud vəziyyəti ilə birlikdə) saxlanılır. 19-cu fəsildə isə tf.keras modelini TensorFlow-un SavedModel formatında necə saxlamaq lazım olduğunu öyrənəcəyik.

Adətən bir skript modeli təlim edib saxlayır, digəri və ya bir neçə skript (və ya veb xidmət) isə həmin modeli yükləyib proqnoz vermək üçün istifadə edir. Modeli yükləmək prosesi isə eyni dərəcədə sadədir:

```
model = keras.models.load_model("my_keras_model.h5")
```

Bu üsul yalnız Silsilə API və ya Funksional API istifadə edildikdə işləyir, lakin təəssüf ki, modelin törəməsi (subklass) zamanı işləməyəcək. Bununla belə, ən azı modelin parametrlərini (çəki və sürüşmələri) saxlamaq və bərpa etmək üçün *save_weights()* və *load_weights()* metodlarından istifadə edə bilərsiniz.Amma bu halda hərşeyi özünüz saxlayıb bərpa etməli olacaqsınız.

Bəs əgər təlim bir neçə saat çəkirsə, nə etməli? Bu, xüsusilə böyük verilənlərlə işləyərkən çox yayılmış haldır. Belə vəziyyətdə modeli yalnız təlimin sonunda saxlamaq kifayət etmir — müntəzəm aralıqlarla modeli saxlama nöqtələri(checkpoint) yaratmaq lazımdır. Bu, kompüterinizin çökməsi və ya prosesi dayandırmanız halında bütün təlim nəticələrini itirməmək

üçün vacibdir. Bəs *fit()* metoduna necə bildirək ki, model hər dəfə avtomatik checkpoint yaratsın? Bunun üçün geri çağırış (callbacks) istifadə olunur.

Geri Çağırışın (Callbacks) istifadəsi

fit() metodu callbacks adlı arqument qəbul edir - bu, obyektlər siyahısıdır ki, Keras onları təlimin başlanğıcında və sonunda, hər bir epoxanın əvvəlində və sonunda, hətta hər dəstin(batch) əvvəlində və sonunda çağırır. Məsələn, *ModelCheckpoint callback* modeli təlim zamanı müntəzəm intervalda adətən hər epoxanın sonunda avtomatik olaraq saxlayır.

```
[...] # build and compile the model
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5")
history = model.fit(X_train, y_train, epochs=10, callbacks=[checkpoint_cb])
```

Üstəlik, əgər təlim zamanı validasiya (yoxlama) dəsti istifadə edirsinizsə, *ModelCheckpoint* obyektini yaradarkən *save_best_only=True* parametrini təyin edə bilərsiniz. Bu halda, model yalnız validasiya dəstindəki nəticə indiyə qədərki ən yaxşı nəticə olduqda saxlanacaq. Bu üsul zamanı, modelin uzun müddət öyrənməsi və overfitting etməsi barədə narahat olmağa ehtiyac yoxdur — sadəcə təlim bitdikdən sonra ən yaxşı nəticəni verən sonuncu modeli bərpa edirsiniz, və bu model validasiya dəsti üzrə ən yaxşı performansa malik olur.

Aşağıdakı kod isə erkən dayandırma üsulunun sadə bir tətbiqidir (bu üsul 4-cü fəsildə təqdim edilmişdi):

```
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5",
                                                save_best_only=True)
history = model.fit(X_train, y_train, epochs=10,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb])
model = keras.models.load_model("my_keras_model.h5") # roll back to best model
```

Erkən dayandırmanı tətbiq etməyin başqa bir yolu isə sadəcə *EarlyStopping callback*-dən istifadə etməkdir. Bu geri çağırış (callback), validasiya dəsti üzərində müəyyən sayda epoxa boyunca heç bir irəliləyiş olmadığını (patience arqumenti ilə təyin edilir) ölçdükdə təlimi dayandırır və istəyə bağlı olaraq ən yaxşı modelə geri dönə biləcək. Həmçinin, bu callback-ləri birləşdirərək həm modelinizin checkpoint-lərini saxlaya bilərsiniz (kompüter çöksə belə) ,həm də irəliləyiş olmadıqda təlimi erkən dayandıra bilərsiniz (vaxt və resursların boşuna sərf olunmasının qarşısını almaq üçün).

```
early_stopping_cb = keras.callbacks.EarlyStopping(patience=10,
                                                  restore_best_weights=True)
history = model.fit(X_train, y_train, epochs=100,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb, early_stopping_cb])
```

Epoxa sayını böyük təyin etmək olar, çünki təlim (training) irəliləyiş olmadıqda avtomatik dayandırılacaq.Bu halda, saxlanmış ən yaxşı modeli ayrıca bərpa etməyə ehtiyac yoxdur, çünki *EarlyStopping callback* ən yaxşı çəkiləri izləyir və təlimin sonunda onları avtomatik bərpa edir.

keras.callbacks paketində çoxsaylı digər geri çağırışlar (callback) də mövcuddur.

Əgər əlavə nəzarətə ehtiyacınız varsa, öz xüsusi geri çağırışlarınızı(callback)yazmaq da mümkündür.Məsələn, aşağıdakı custom callback təlim zamanı validasiya itkisi ilə təlim itkisinin nisbətini göstərir (məsələn, modelin əzbərləməsini(overfitting) aşkarlamaq üçün):

```
class PrintValTrainRatioCallback(keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs):
        print("\nval/train: {:.2f}".format(logs["val_loss"] / logs["loss"]))
```

Təxmin etdiyiniz kimi *siz on_train_begin()*, *on_train_end()*, *on_epoch_begin()*, *on_epoch_end()*, *on_batch_begin()*, *on_batch_end()* tətbiq edə bilərsiz. Geri çağırışlar (Callback) həmçinin qiymətləndirmə və proqnozlar zamanı lazım olduqda da (məsələn debug zamanı) istifadə edilə bilər. Qiymətləndirmə *zamanı on_test_begin()*, *on_test_end()*, *on_test_batch_begin()*, *on_test_batch_end()* (bu metodlar *evaluate()* tərəfindən çağırılır),proqnoz zamanı *on_predict_begin()*, *on_predict_end()*, *on_predict_batch_begin()* və *ya on_predict_batch_end()* (bu metodlar *predict()* tərəfindən çağırılır) tətbiq edilməlidir.

İndi isə tf.keras istifadə edərkən mütləq sahib olmalı olduğunuz bir vasitəyə baxaq: TensorBoard.

TensorBoard ilə Vizualizasiya

TensorBoard interaktiv vizualizasiya üçün əla bir vasitədir. Onu istifadə edərək təlim zamanı öyrənmə əyrilərini izləyə,bir neçə təlim icrasının öyrənmə əyrilərini müqayisə edə, hesablama qrafikini vizuallaşdıra,təlim statistikalarını analiz edə, model tərəfindən yaradılmış şəkilləri göstərə, çoxölçülü məlumatları 3D-ə proyeksiya edib avtomatik klasterləşdirə və s. edə bilərsiniz.

TensorBoard, TensorFlow quraşdırıldıqda avtomatik olaraq gəlir, yəni artıq sizdə mövcuddur.

TensorBoard istifadə etmək üçün proqramınızı belə dəyişdirməlisiniz ki, vizuallaşdırmaq istədiyiniz məlumatları xüsusi ikili jurnal (log) fayllarına yazsın hansı ki, *event faylları* adlanır.Hər bir ikili məlumat qeydiyyatı *summary* adlanır. TensorBoard serveri log qovluğunu(directory) izləyir və dəyişiklikləri avtomatik olaraq vizuallaşdırmalara əlavə edir: bu sayədə, təlim zamanı öyrənmə əyriləri kimi canlı datanı vizuallaşdırmaq mümkün olur (qısa gecikmə ilə). Ümumiyyətlə, TensorBoard serverini əsas jurnal (log) qovluğuna yönləndirmək və proqramınızı hər dəfə işə düşəndə fərqli bir alt qovluğa yazacaq şəkildə tənzimləmək lazımdır. Bu yanaşma ilə

eyni TensorBoard server nümunəsi vasitəsilə bir birinə qarışmadan çoxlu icra nəticələrini (multiple runs) vizuallaşdıra və müqayisə edə bilərsiniz.

Gəlin TensorBoard qeydlərimiz üçün istifadə edəcəyimiz əsas jurnal (log) qovluğunu müəyyən etməklə başlayaq. Daha sonra hər bir icra zamanı fərqli olması üçün cari tarix və vaxta əsaslanaraq alt qovluq yolunu yaradan kiçik bir funksiya tərtib edəcəyik. Log qovluğunun adına baxdığınızda TensorBoard-da nəyi izlədiyinizi daha asan başa düşmək üçün, məsələn, test etdiyiniz hiperparametr dəyərləri kimi əlavə məlumatları da ada daxil edə bilərsiniz:

```
import os
root_logdir = os.path.join(os.curdir, "my_logs")

def get_run_logdir():
    import time
    run_id = time.strftime("run_%Y_%m_%d-%H_%M_%S")
    return os.path.join(root_logdir, run_id)

run_logdir = get_run_logdir() # e.g., './my_logs/run_2019_06_07-15_15_22'
```

Yaxşı xəbər odur ki, Keras hazır bir TensorBoard() callback təmin edir:

```
[...] # Build and compile your model
tensorboard_cb = keras.callbacks.TensorBoard(run_logdir)
history = model.fit(X_train, y_train, epochs=30,
                    validation_data=(X_valid, y_valid),
                    callbacks=[tensorboard_cb])
```

Və işin əsası bu qədərdir! Onu istifadə etmək bundan asan ola bilməzdi.Əgər bu kodu işə salsanız, *TensorBoard() callback* log qovluğunu (lazım olarsa əsas qovluqlar(parent directory) ilə birlikdə) və təlim zamanı event fayllarını yaradacaq və onlara icmal (summary) yazacaq. Proqramı ikinci dəfə işə saldıqdan sonra (çox güman ki, bəzi hyperparameterləri dəyişdirdikdən sonra) sizdə təxminən belə bir qovluqlar strukturu yaranacaq:

```
my_logs/
├── run_2019_06_07-15_15_22
│   ├── train
│   │   ├── events.out.tfevents.1559891732.mycomputer.local.38511.694049.v2
│   │   ├── events.out.tfevents.1559891732.mycomputer.local.profile-empty
│   │   └── plugins/profile/2019-06-07_15-15-32
│   │       └── local.trace
│   └── validation
│       └── events.out.tfevents.1559891733.mycomputer.local.38511.696430.v2
└── run_2019_06_07-15_15_49
    └── [...]
```

Hər icrada bir qovluq olur hansı ki,hər biri təlim logları və yoxlama (validation) logları üçün bir altqovluqdan ibarət olur. Hər iki altqovluqda event faylları mövcuddur, lakin təlim loglarına əlavə

olaraq profil izləmə də daxil olur: Bu, TensorBoard-un modelin hər hissəsində nə qədər vaxt sərf olunduğunu göstərməsinə imkan verir və performansın dar boğazlarını tapmaq üçün çox faydalıdır.

Sonrakı addım, TensorBoard serverini işə salmaqdır. Bunu etməyin bir yolu terminalda əmri icra etməkdir. Əgər TensorFlow-nu virtualenv daxilində quraşdırmısınızsa, onu aktivləşdirin. Sonra layihənin əsas qovluğundan (və ya uyğun log qovluğuna istinad edərək istənilən yerdən) aşağıdakı əmri işlədin:

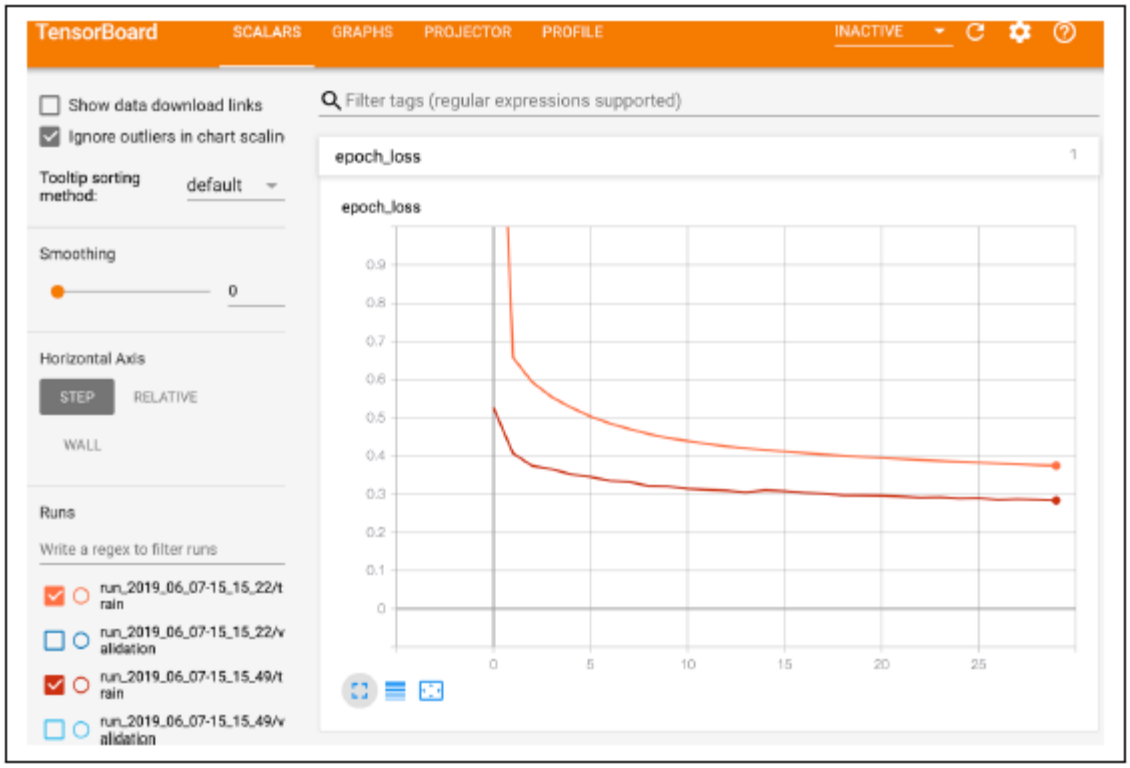
```
$ tensorboard --logdir=./my_logs --port=6006
TensorBoard 2.0.0 at http://mycomputer.local:6006/ (Press CTRL+C to quit)
```

Əgər *tensorboard* skriptini tapa bilmirsizsə, o zaman *PATH* mühit dəyişənini (environment variable) elə yeniləməlisiniz ki, bu dəyişən skriptin quraşdırıldığı qovluğu da özündə əks etdirsin(alternativ olaraq, əmr sətrində tensorboard əvəzinə *python3 -m tensorboard.main* yazmaqla da onu işə sala bilərsiniz). Server işə düşdükdən sonra, veb-brauzeri açaraq http://localhost:6006 ünvanına daxil olmaq mümkündür.

Alternativ olaraq, TensorBoard-u birbaşa Jupyter-də istifadə edə bilərsiniz. Bunun üçün aşağıdakı əmrləri işlədin. İlk sətr TensorBoard uzantısını yükləyir, ikinci sətr isə TensorBoard serverini port 6006-da işə salır (əgər artıq işə düşməyibsə) və ona qoşulur.

```
%load_ext tensorboard
%tensorboard --logdir=./my_logs --port=6006
```

Hər iki halda da TensorBoard-un veb interfeysini görməlisiniz. *SCALARS* tabına klikləyərək öyrənmə əyrilərini izləyə bilərsiniz (bax: Şəkil 10-17). Sol alt küncdə vizuallaşdırmaq istədiyiniz qovluqları seçin (məsələn, birinci və ikinci icranın təlim qovluqları) və *epoch_loss* -a klikləyin. Görəcəksiz ki, hər iki icra zamanı təlim itkisi yaxşı şəkildə azalıb, lakin ikinci icrada daha sürətlə aşağı düşmə baş verib. Bunun səbəbi odur ki, ikinci icra üçün öyrənmə əyrisi 0.001 əvəzinə 0.05 (*optimizer=keras.optimizers.SGD(lr=0.05))* istifadə olunmuşdu,



Şəkil 10-17. TensorBoard ilə öyrənmə əyrilərinin vizuallaşdırılması

Siz həmçinin bütün qrafiki, öyrənilmiş çəkiləri (learned weights) (3D-ə proyeksiya edilmiş) və ya profil izləməni vizuallaşdıra bilərsiniz. TensorBoard() callback əlavə məlumatları da qeyd etmək (to log) üçün seçimlər təklif edir, məsələn, yerləşdirmə(embeddings) (bax: Chapter 13).

Bundan əlavə, TensorFlow aşağı səviyyəli API təqdim edir: *tf.summary* paketi. Bu API ilə, *create_file_writer()* funksiyasından istifadə edərək *SummaryWriter* yarada və onu konteks olaraq istifadə edərək sklayarlar histoqramlar,şəkillər, səs və yazı kimi məlumatları yaza(to log) bilərsiniz. Bu məlumatları sonra TensorBoard vasitəsilə vizuallaşdırmaq mümkündür.

```
test_logdir = get_run_logdir()
writer = tf.summary.create_file_writer(test_logdir)
with writer.as_default():
    for step in range(1, 1000 + 1):
        tf.summary.scalar("my_scalar", np.sin(step / 10), step=step)
        data = (np.random.randn(100) + 2) * step / 100 # some random data
        tf.summary.histogram("my_hist", data, buckets=50, step=step)
        images = np.random.rand(2, 32, 32, 3) # random 32×32 RGB images
        tf.summary.image("my_images", images * step / 1000, step=step)
        texts = ["The step is " + str(step), "Its square is " + str(step**2)]
        tf.summary.text("my_text", texts, step=step)
        sine_wave = tf.math.sin(tf.range(12000) / 48000 * 2 * np.pi * step)
        audio = tf.reshape(tf.cast(sine_wave, tf.float32), [1, -1, 1])
        tf.summary.audio("my_audio", audio, sample_rate=48000, step=step)
```

Bu, əslində TensorFlow və Deep Learning-dən kənarda da faydalı ola biləcək bir vizualizasiya vasitəsidir.

Gəlin bu fəsildə öyrəndiklərimizi ümumiləşdirək: Neyral netlərin mənşəyini gördük, MLP nədir və onu klassifikasiya və reqressiya üçün onu necə istifadə etmək olar öyrəndik, tf.keras-ın Silsilə API-sindən istifadə edərək MLP-lər yaratmağı öyrəndik, daha mürəkkəb model arxitekturaları yaratmaq üçün Funksional API və ya Subklass API-dən istifadə etməyi öyrəndik, Modeli saxlamağı və yenidən başlatmağı, saxlama nöqtələri(checkpoint) üçün geri çağırışlardan (callbacks) istifadə etməyi ,erkən dayandırma və digər xüsusiyyətləri istifadə etməyi öyrəndik. Nəhayət, vizualizasiya üçün TensorBoard-dan istifadə etməyi öyrəndik. Artıq neyral şəbəkələri müxtəlif problemləri həll etmək üçün istifadə edə bilərsiniz! Lakin siz gizli qatların sayı, neyron sayı və digər hiperparametrləri necə müəyyən etməyi düşünə bilərsiz. Gəlin bunu indi nəzərdən keçirək.

Neyral şəbəkə hiperparametrlərinin incə tənzimlənməsi(fine-tuning)

Neyral şəbəkələrinin çevikliyi eyni zamanda onların əsas çatışmazlıqlarından biridir: tənzimlənməsi gərəkən çox sayda hiperparametr mövcuddur. Siz yalnız hər hansı təsəvvür edilə bilən şəbəkə arxitekturasını istifadə zamanı deyil, hətta sadə bir MLP-də (Çoxqatlı Perseptron) belə qatların sayı, hər qatda neyronların miqdarı, hər qatda istifadə olunan aktivasiya funksiyasının tipi, çəki (weight) başlanğıc dəyərlərinin təyin olunma məntiqi və daha bir çox parameter dəyişdirilə bilər.Bəs hansı hiperparametr kombinasiyasının sizin tapşırığınız üçün ən optimal kombinasiya olduğunu necə müəyyən etmək olar?

Bir üsul sadəcə müxtəlif hiperparametr kombinasiyalarını sınaqdan keçirmək və validasiya(yoxlama) dəsti üzərində ən yaxşı nəticə verənini seçməkdir (və ya K-fold cross-validation üsulundan istifadə etmək olar). Məsələn, 2-ci fəsildə olduğu kimi, hiperparametrləri araşdırmaq üçün GridSearchCV və ya RandomizedSearchCV alətlərindən istifadə edə bilərik.

Bunun üçün Keras modellərimizi Scikit-Learn-dəki adi reqressor obyektlərinə bənzər formaya salmaq lazımdır. İlk addım verilmiş hiperparametrlər toplusuna əsasən Keras modelini quran(build) və tərtib(compile) edən bir funksiya yaratmaqdır.

```
def build_model(n_hidden=1, n_neurons=30, learning_rate=3e-3, input_shape=[8]):
    model = keras.models.Sequential()
    model.add(keras.layers.InputLayer(input_shape=input_shape))
    for layer in range(n_hidden):
        model.add(keras.layers.Dense(n_neurons, activation="relu"))
    model.add(keras.layers.Dense(1))
    optimizer = keras.optimizers.SGD(lr=learning_rate)
    model.compile(loss="mse", optimizer=optimizer)
    return model
```

Bu funksiya bir dəyişənli reqressiya üçün sadə bir Silsilə model yaradır (yəni outputda yalnız bir neyron olur). Bu model istifadəçidən alınan giriş ölçüsü(input shape), gizli qatların sayı və neyronların miqdarı əsasında qurulur və təyin olunmuş öyrənmə dərəcəsi ilə konfiqurasiya edilmiş SGD (Stochastic Gradient Descent) optimizatoru vasitəsilə kompilyasiya(modelin öyrənməyə hazırlanması) edilir. Yaxşı bir təcrübə kimi, Scikit-Learn kitabxanasında olduğu kimi, mümkün qədər çox hiperparametr üçün məntiqli default dəyərlər təyin etmək tövsiyə olunur. Növbəti addımda isə, bu *build_model()* funksiyasına əsaslanaraq bir *KerasRegressor* yaradaq:

```
keras_reg = keras.wrappers.scikit_learn.KerasRegressor(build_model)
```

KerasRegressor obyekti, *build_model()* funksiyası vasitəsilə qurulan Keras modelinin ətrafında olan nazik bükücü (thin wrapper) rolunu oynayır. Model yaradılarkən hər hansı hiperparametr təyin etmədiyimiz üçün, o, *build_model()* funksiyasında müəyyən edilmiş standart hiperparametrləri istifadə edəcək. İndi bu obyekti adi bir Scikit-Learn reqressoru kimi istifadə edə bilərik: onu *fit()* metodu ilə öyrətmək(training), *score()* metodu ilə qiymətləndirmək və *predict()* metodu vasitəsilə ona proqnozlar verdirmək olar. Aşağıdakı kodda gördüyümüz kimi:

```
keras_reg.fit(X_train, y_train, epochs=100,
              validation_data=(X_valid, y_valid),
              callbacks=[keras.callbacks.EarlyStopping(patience=10)])
mse_test = keras_reg.score(X_test, y_test)
y_pred = keras_reg.predict(X_new)
```

Qeyd etmək lazımdır ki, *fit()* metoduna ötürülən bütün əlavə parametrlər birbaşa əsas Keras modelinə göndəriləcək. Həmçinin nəzərə almaq lazımdır ki, əldə edilən skor(score) dəyəri orta kvadratik xətanın (MSE) əks nəticəsini göstərəcək, çünki Scikit-Learn skor(score) istəyir, itki deyil ("yüksək nəticə daha yaxşıdır").

Məqsədimiz yalnız bir modeli öyrətmək və qiymətləndirmək deyil yüzlərlə fərqli model variantını sınaqdan keçirib, validasiya dəsti üzərində ən yaxşı nəticə verəni tapmaqdır. Çoxlu sayda hiperparametr olduğuna görə, Grid search yerinə Randomized search metodundan istifadə etmək daha məqsədəuyğundur (bu mövzunu 2-ci fəsildə müzakirə etmişdik).

Gəlin indi gizli qatların sayı, neyronların miqdarı və öyrənmə dərəcəsini araşdıraq.

```
from scipy.stats import reciprocal
from sklearn.model_selection import RandomizedSearchCV

param_distribs = {
    "n_hidden": [0, 1, 2, 3],
    "n_neurons": np.arange(1, 100),
    "learning_rate": reciprocal(3e-4, 3e-2),
}

rnd_search_cv = RandomizedSearchCV(keras_reg, param_distribs, n_iter=10, cv=3)
rnd_search_cv.fit(X_train, y_train, epochs=100,
                  validation_data=(X_valid, y_valid),
                  callbacks=[keras.callbacks.EarlyStopping(patience=10)])
```

Bu addım, 2-ci fəsildə etdiklərimizlə demək olar ki, eynidir. Yeganə fərq ondadır ki, burada *fit()* metoduna əlavə parametrlər ötürülür və bu parametrlər əsas Keras modellərinə yönləndirilir. Qeyd etmək vacibdir ki, *RandomizedSearchCV* metodu *K-fold cross-validation* üsulundan istifadə edir; bu səbəbdən *X_valid* və *y_valid* istifadə etmir, çünki yalnız təlimin vaxtından əvvəl dayandırılması üçün istifadə olunur.

Axtarış prosesi aparat təminatından (hardware), datanın ölçüsündən, modelin mürəkkəbliyindən və *n_iter* ilə *cv* parametrlərinin dəyərlərindən asılı olaraq bir neçə saat davam edə bilər.Axtarış başa çatdıqdan sonra isə, tapılmış ən yaxşı hiperparametrlərə, ən yaxşı skora və öyrədilmiş Keras modelinə aşağıdakı kimi daxil olmaq mümkündür:

```
>>> rnd_search_cv.best_params_
{'learning_rate': 0.0033625641252688094, 'n_hidden': 2, 'n_neurons': 42}
>>> rnd_search_cv.best_score_
-0.3189529188278931
>>> model = rnd_search_cv.best_estimator_.model
```

İndi siz bu modeli yadda saxlaya, test dəsti üzərində qiymətləndirə və əgər nəticələr sizi qane edirsə, istehsal mühitinə tətbiq edə bilərsiniz. Randomized search(təsadüfi axtarış) üsulundan istifadə etmək çətin deyil və o, bir çox nisbətən sadə problemlər üçün uğurla işləyir. Lakin təlim prosesi ləng gedirsə (məsələn, böyük məlumat dəstləri və ya mürəkkəb modellər üçün), bu yanaşma hiperparametr məkanının çox kiçik hissəsini araşdıra bilir. Bu problemi axtarışı müəyyən qədər manual şəkildə etməklə yüngülləşdirmək mümkündür: əvvəlcə hiperparametrlərin geniş dəyərlər aralığında sürətli bir təsadüfi axtarış aparın, sonra isə birinci mərhələdə tapılmış ən yaxşı nəticələrin ətrafında daha kiçik aralıqlı ikinci axtarış həyata keçirin və s. Bu yanaşma hipeerparameterlerin optimal toplusunu önə çıxara biləcək. Lakin bu yanaşma çox vaxt aparır və resurs baxımından səmərəli hesab edilmir.

Xoşbəxtlikdən, hiperparametr məkanını təsadüfi deyil, daha səmərəli şəkildə araşdırmağa imkan verən bir çox texnikalar mövcuddur. Bu metodların əsas ideyası sadədir: əgər məkanın hər hansı bir sahəsi yaxşı nəticə verirsə, o sahə daha dərindən araşdırılmalıdır. Bu texnikalar “zooming”

(böyütmə) prosesini avtomatik həyata keçirir və daha qısa müddətdə daha yaxşı nəticələr əldə etməyə imkan yaradır. Aşağıda hiperparametrlərin optimallaşdırılması üçün istifadə oluna bilən bəzi Python kitabxanaları mövcuddur:

Hyperopt

Bütün növ mürəkkəb axtarış məkanlarında(öyrənmə dərəcəsi kimi real dəyərlər və qatların sayı kimi diskret dəyərlər daxil olmaqla) optimallaşdırma aparmaq üçün geniş istifadə olunan məşhur kitabxanadır.

Hyperas, **kopt** və ya **Talos**

Keras modelləri üçün hiperparametr optimallaşdırmasını asanlaşdıran kitabxanalar (İlk ikisi Hyperopt əsasında qurulub).

Keras Tuner

Google tərəfindən hazırlanmış, Keras modelləri üçün sadə və rahat hiperparametr optimallaşdırma kitabxanasıdır. Vizualizasiya və analiz üçün ayrıca yerləşdirilmiş xidmət (hosted service) təqdim edir.

Scikit-Optimize (skopt)

Ümumi məqsədli optimallaşdırma kitabxanasıdır. Onun *BayesSearchCV* sinfi Bayesian optimallaşdırma aparır və *GridSearchCV*-yə bənzər interfeysə malikdir.

Spearmint

Bayesian optimallaşdırma kitabxanasıdır.

Hyperband

Hyperband, Lisha Li və həmkarlarının son dövrlərdə dərc edilmiş "Hyperband"[22] adlı elmi məqaləsinə əsaslanan, hiperparametrlərin sürətli tənzimlənməsi üçün nəzərdə tutulmuş kitabxanadır.

Sklearn-Deap

Sklearn-Deap, təkamül alqoritmləri əsasında işləyən hiperparametr optimallaşdırma kitabxanasıdır və *GridSearchCV*-yə bənzər interfeysə malikdir.

Bundan əlavə, bir çox şirkətlər hiperparametrlərin optimallaşdırılması üçün xidmətlər də təqdim edir. Bu barədə 19-cu fəsildə, Google Cloud AI Platform-un hiperparametr tənzimləmə xidmətində müzakirə edəcəyik. Digər alternativlərə Arimo, SigOpt və CallDesk-in Oscar xidmətləri daxildir.

Hiperparametr tənzimlənməsi hələ də fəal tədqiqat sahəsidir, və təkamül alqoritmləri son illərdə yenidən populyarlıq qazanmaqdadır. Məsələn, DeepMind tərəfindən 2017-ci ildə nəşr edilmiş

[22] Lisha Li və digərləri, "Hyperband: Hiperparametr Optimallaşdırmasına Yenilikçi Bandit-əsaslı Yanaşma," *Journal of Machine Learning Research* 18 (Aprel 2018): 1–52.

məqaləyə[23] göz atın. Burada müəlliflər model populyasiyasını və onların hiperparametrlərini birlikdə optimallaşdırmaq metodunu təqdim ediblər. Google da təkamül yanaşmasını təkcə hiperparametrlərin deyil, həm də neyral şəbəkə arxitekturasının optimal formasını tapmaq üçün istifadə etmişdir. Onların AutoML adlı alətlər toplusu artıq bulud xidməti kimi mövcuddur. Bəlkə də yaxın gələcəkdə neyral şəbəkələrin manual qurulması dövrü başa çatacaq. Bu mövzuda Google-un paylaşdığı bloq yazısına baxmaq maraqlı ola bilər. Əslində, təkamül alqoritmləri artıq individual neyral şəbəkələrin öyrədilməsi prosesində də Qradient Enmə üsulunun alternativi kimi uğurla tətbiq edilmişdir. Məsələn, Uber-in 2017-ci ildə dərc etdiyi məqalədə müəlliflər Dərin Neyrotəkamül adlı texnikanı təqdim edirlər.

Lakin bütün bu yeniliklərə, alətlərə və xidmətlərə baxmayaraq, yenə də hər bir hiperparametr üçün münasib dəyərlərin diapazonu haqqında ilkin anlayışa malik olmaq kömək edir. Belə ki siz həm sürətli prototiplər qura, həm də axtarış məkanını məhdudlaşdıra bilərsiniz. Aşağıdakı bölmələr MLP-də gizli qatların və neyronların sayını seçmək, eləcə də bəzi əsas hiperparametrlər üçün yaxşı dəyərləri təyin etmək üzrə təlimatlar təqdim edir.

Gizli Qatların Sayı

Bir çox tapşırıqlar üçün bir gizli qatdan başlaya bilər və yaxşı nəticələr əldə edə bilərsən. Teorik olaraq, yalnız bir gizli qata malik bir MLP, kifayət qədər çox neyron olduqda, istənilən mürəkkəb funksiyanı modelləşdirə bilər. Lakin mürəkkəb problemlərdə dərin şəbəkələr (deep networks), səthi şəbəkələrə nisbətən daha çox *parametr səmərəliliyinə* malikdir: onlar mürəkkəb funksiyaları səthi netlərdən çox çox az neyron vasitəsilə modelləşdirə bilir və eyni miqdarda təlim məlumatı ilə daha yüksək performans əldə edirlər.

Bu fərqi anlamaq üçün fərz edək ki, sizdən software ilə bir meşə şəkli çəkmək tələb olunur, lakin nüsxələmək və yapışdırmaq funksiyasından istifadə etmək qadağandır. Bu halda hər bir ağacı, budağı,yarpağı ayrı-ayrılıqda çəkməli olacaqsınız və bu çox vaxt aparacaq. Əgər siz bir yarpaq çəkib onu nüsxələyərək budaqlar, sonra bu budaqlardan ağaclar, və nəhayət bu ağaclardan meşə yarada bilsəniz, işiniz çox tez bitərdi. Real datalar da çox vaxt belə iyerarxik şəkildə qurulur və dərin neyral şəbəkələr bu xüsusiyyətdən avtomatik şəkildə istifadə edir: Aşağı gizli qatlar aşağı səviyyəli strukturları (məsələn, müxtəlif istiqamətli və formalı xətt seqmentləri) modelləşdirir, Orta gizli qatlar bu aşağı səviyyəli strukturları birləşdirərək orta səviyyəli strukturları(məsələn, dairələr, kvadratlar) modelləşdirir, Ən yüksək gizli qatlar və çıxış qatı(output layer) bu orta səviyyəli strukturları birləşdirərək yüksək səviyyəli strukturları(məsələn, üzləri) modelləşdirir.

Bu iyerarxik arxitektura təkcə dərin şəbəkələrin yaxşı həllə daha tez yaxınlaşmasına kömək etmir, həm də onların yeni datasetləri ümumiləşdirmə qabiliyyətini inkişaf etdirir.Məsələn, əgər siz artıq üz tanıma üzrə öyrədilmiş bir modelə maliksinizsə və indi saç modellərini tanıyan yeni bir şəbəkə öyrətmək istəyirsinizsə, ilk modelin aşağı qatlarını təkrar istifadə edərək təlim prosesinə başlaya

[23] Max Jaderberg və digərləri, "Neyron Şəbəkələrinin Populyasiya-əsaslı Təlimi," *arXiv preprint* arXiv:1711.09846 (2017).

bilərsiniz.Yeni modelin ilkin bir neçə qatının çəki və sürüşmələrini(bias) təsadüfi şəkildə təyin etmək əvəzinə, onları ilk modelin uyğun qatlarının dəyərləri ilə başlatmaq mümkündür. Bu halda, yeni şəbəkə əksər şəkillərdə olan aşağı səviyyəli vizual xüsusiyyətləri yenidən sıfırdan öyrənməyə ehtiyac duymayacaq, yalnız yüksək səviyyəli xüsusiyyətləri (məsələn, saç modellərini) öyrənəcək. Bu yanaşma *bilginin ötürülməsi (transfer learning)* adlanır.

Yekun olaraq, bir çox problemlər üçün sadəcə bir və ya iki gizli qatdan ibarət şəbəkə ilə başlaya bilərsən və neyrol şəbəkə yaxşı işləyəcək. Məsələn: MNIST datasetində yalnız bir gizli qatda bir neçə yüz neyronla 97%-dən yuxarı dəqiqlik əldə etmək mümkündür; İki gizli qatda eyni sayda neyronla bu göstərici 98%-ə çata bilər, və təlim vaxtı təxminən eyni qalır. Daha mürəkkəb problemlərdə isə, overfitting yaranana qədər qatların sayını artırmaq olar. Məsələn, böyük şəkil klassifikasiyası və ya nitq tanıma kimi mürəkkəb tapşırıqlar adətən onlarla (bəzən yüzlərlə) qata malik şəbəkələr (lakin bu qatlar tam əlaqəli olmur — bu barədə 14-cü fəsildə danışılacaq) və böyük miqdarda öyrənmə datası tələb edir. Belə şəbəkələri sıfırdan öyrətməyə çox vaxt ehtiyac olmur: adətən oxşar tapşırıq üçün əvvəlcədən öyrədilmiş bir modelin hissələrindən istifadə edilir. Bu, həm təlim müddətini azaldır, həm də daha az data tələb olunur.(bu barədə daha ətraflı 11-ci fəsildə danışılacaq).

Hər Gizli qatdakı neyronların sayı

Input və output qatlarındakı neyronların sayı tapşırığın tələb etdiyi input və output növü ilə müəyyən olunur. Məsələn, MNIST tapşırığı 28 × 28 = 784 input neyronu və 10 output neyronu tələb edir.

Gizli qatlara gəlincə, əvvəllər onları piramida formasında tərtib etmək geniş yayılmış bir yanaşma idi, yəni hər qatda getdikcə daha az neyron yerləşdirilirdi — bunun məntiqi isə belə idi ki, çoxsaylı aşağı səviyyəli xüsusiyyətlər (feature) daha az sayda yüksək səviyyəli xüsusiyyətlərə birləşə bilər. MNIST üçün tipik bir neyron şəbəkəsi üç gizli qatdan ibarət ola bilərdi: birincidə 300 neyron, ikincidə 200, üçüncüdə isə 100 neyron. Lakin bu praktika artıq demək olar ki, tərk edilmişdir, çünki əksər hallarda bütün gizli qatlarda eyni sayda neyron istifadə etmək ya eyni nəticəni verir, ya da daha yaxşı nəticə göstərir; üstəlik, hər qat üçün bir hiperparametr tənzimlənməsi əvəzinə yalnız bir hiperparametrin tənzimlənməsi kifayət edir. Bununla belə, bəzi hallarda verilənlər toplusundan asılı olaraq birinci gizli qatın digər qatlara nisbətən daha böyük olması faydalı ola bilər.

Qatların sayı kimi, neyronların sayını da şəbəkə overfitting etməyə başlayana qədər tədricən artırmaq olar. Lakin praktik baxımdan çox vaxt ehtiyac duyduğundan bir qədər artıq qat və neyron seçmək, sonra isə erkən dayandırma (early stopping) və digər regulyarizasiya texnikalarından istifadə etmək overfitting-in qarşısını almaq daha sadə və səmərəlidir. Google alimlərindən Vincent Vanhoucke bu yanaşmanı "stretch pants" (elastik şalvar) yanaşması adlandırmışdır: mükəmməl ölçüdə şalvar tapmaq üçün vaxt itirməkdənsə, sadəcə olaraq elastik şalvar geyinmək daha rahatdır belə ki, o, özü uyğun ölçüyə qədər yığılacaq. Bu yanaşma ilə siz modelin performansını azalda biləcək dar boğaz qatlarından qaça bilərsiniz. Digər tərəfdən, əgər qat çox az

neyrona malik olarsa, bu qat input məlumatından bütün faydalı informasiyanı saxlamaq üçün kifayət qədər təsvir gücünə malik olmayacaq. (Məsələn, iki neyronu olan bir qatdan yalnız ikiölçülü (2D) məlumat çıxış edə bilər, buna görə də üçölçülü (3D) məlumatı emal edərsə müəyyən məlumat itkisi baş verəcəkdir). Şəbəkənin qalan hissəsi nə qədər böyük və güclü olsa da, bu itirilmiş məlumat bərpa oluna bilməz.

Ümumilikdə, hər qatda neyronların sayını artırmaq əvəzinə qatların sayını artırmaqla daha yüksək səmərəlilik əldə edəcəksiniz.

Öyrənmə dərəcəsi, Dəst (Batch) Ölçüsü və Digər Hiperparametrlər

Gizli qatların və neyronların sayı MLP-də tənzimləyə biləcəyiniz yeganə hiperparametrlər deyildir. Aşağıda ən vacib hiperparametrlər və onların necə təyin ediləcəyi ilə bağlı bəzi tövsiyələr verilmişdir:

Öyrənmə dərəcəsi

Öyrənmə dərəcəsi şübhəsiz ki, ən vacib hiperparametrlərdən biridir. Ümumiyyətlə, optimal öyrənmə dərəcəsi maksimal öyrənmə dərəcəsinin təxminən yarısıdır (Təlim alqoritminin divergensiya etdiyi(uzaqlaşma) öyrənmə dərəcəsindən yuxarı olan dəyər, 4-cü fəsildə gördüyümüz kimi).Yaxşı öyrənmə dərəcəsini tapmağın bir yolu budur: modeli bir neçə yüz iterasiya üçün öyrətmək, çox aşağı öyrənmə dərəcəsindən başlamaq (məsələn, 10^{-5}) və onu tədricən çox yüksək bir dəyərə (məsələn, 10) qədər artırmaq. Bu, hər iterasiyada öyrənmə dərəcəsini sabit bir faktorla vurmaqla edilir (məsələn, 500 iterasiya ərzində 10^{-5}-dən 10-a keçmək üçün $\exp(\log(10^{6})/500)$ ilə). Əgər öyrənmə dərəcəsinin(öyrənmə dərəcəsi üçün log miqyasını istifadə etməklə) funksiyası kimi itkini (loss) vizuallaşdırsanız, əvvəlcə onun azaldığını görəcəksiniz. Lakin bir müddət sonra öyrənmə dərəcəsi çox yüksək olacaq və itki yenidən artacaq: optimal öyrənmə dərəcəsi itkinin yenidən artmağa başladığı nöqtədən bir qədər aşağı olacaq (adətən dönmə nöqtəsindən təxminən 10 dəfə aşağı). Daha sonra modeli yenidən başladıb bu yaxşı öyrənmə dərəcəsi ilə normal şəkildə təlim edə bilərsiniz. Daha ətraflı öyrənmə dərəcəsi texnikalarına 11-ci fəsildə baxacağıq.

Optimizer (Optimallaşdırıcı)

Sadə Mini-dəst Qradient Enmədən daha yaxşı bir optimallaşdırıcı seçmək (və onun hiperparametrlərini tənzimləmək) də olduqca vacibdir. 11-ci fəsildə bir neçə qabaqcıl optimallaşdırıcı ilə tanış olacağıq.

Dəst (Batch) ölçüsü

Dəst ölçüsü modelinizin performansına və təlim müddətinə əhəmiyyətli təsir göstərə bilər. Böyük dəst ölçüləri istifadə etməyin əsas üstünlüyü odur ki, GPU kimi hardware akseleratorları onları effektiv şəkildə işlədə bilir (bax :19-cu fəsil), beləliklə təlim

alqoritmi hər saniyədə daha çox nümunə görəcək. Buna görə bir çox tədqiqatçı və təcrübəçilər GPU RAM-ına uyğun gələn böyük dəst ölçüsü istifadə etməyi tövsiyə edir. Lakin burada bir çatışmazlıq var: praktikada böyük dəst (batch) ölçüləri tez-tez təlimin (training) qeyri-sabit olmasına səbəb olur, xüsusilə təlimin başlanğıcında, və nəticədə əldə olunan model kiçik batch ölçüsü ilə öyrədilmiş model qədər yaxşı ümumiləşdirə bilməyə bilər. 2018-ci ilin aprel ayında Yann LeCun belə bir tvit atmışdı: "Dostlarınızın, mini-dəst ölçülərini 32-dən böyük etməsinə icazə verməyin," və Dominik Masters və Carlo Luşhi tərəfindən 2018-ci ildə aparılmış bir tədqiqatı sitat gətirmişdi[24]; həmin tədqiqat göstərdi ki, kiçik dəstlərdən (2-dən 32-yə qədər) istifadə etmək daha məqsədəuyğundur, çünki kiçik dəstlər daha az təlim müddəti ilə daha yaxşı modellərə gətirib çıxarır. Digər tərəfdən, bəzi tədqiqatlar əksini göstərir: 2017-ci ildə Elad Hoffer və digərlərinin[25], Priya Goyal və digərlərinin[26] məqalələri göstərdi ki, öyrənmə dərəcəsini tədricən artırmaq (yəni, əvvəlcə kiçik öyrənmə dərəcəsi ilə təlimə başlamaq və sonra 11-ci fəsildə görəcəyimiz kimi artırmaq.) kimi müxtəlif texnikalardan istifadə etməklə çox böyük dəst ölçülərindən (8,192-ə qədər) istifadə etmək mümkündür. Bu, heç bir ümumiləşdirmə boşluğuna səbəb olmadan çox qısa təlim(training) müddətinə gətirib çıxardı. Belə ki, bir strtegiya budur ki öyrənmə dərəcəsinin tədrici artımı ilə böyük dəst ölçüsünü istifadə etməyə çalışasınız, Əgər təlim qeyri-sabitdirsə və ya nəticə məyusedicidirsə, kiçik dəst ölçüsünü sınayasınız.

Aktivasiya funksiyası

Bu fəsildə aktivasiya funksiyasını necə seçmək barədə danışdıq: Qısacası, ReLU aktivasiya funksiyası bütün gizli qatlar üçün yaxşı bir standart seçim olacaq. Çıxış qatı üçün isə bu, əslində sizin tapşırığınızdan asılıdır.

İterasiya sayı

Çox hallarda, təlim iterasiyalarının sayını tənzimlənməyə ehtiyac yoxdur: sadəcə erkən dayandırmadan (early stopping) istifadə edin.

Optimal öyrənmə dərəcəsi digər hiperparametrlərdən—xüsusilə dəst ölçüsündən—asılıdır, buna görə hər hansı hiperparametri dəyişdirirsinizsə, öyrənmə dərəcəsini də uyğun şəkildə yenilədiyinizə əmin olun.

[24] Dominic Masters və Carlo Luschi, "Dərin Neyron Şəbəkələri üçün Kiçik Dəst Təliminə Yenidən Baxış", *arXiv* ilkin nəşri, arXiv:1804.07612 (2018).
[25] Elad Hoffer və digərləri, "Daha Uzun Təlim, Daha Yaxşı Ümumiləşdirmə: Neyron Şəbəkələrin Böyük Dəst Təlimində Ümumiləşdirmə Fərqinin Aradan Qaldırılması", *31-ci Beynəlxalq Neyron İnformasiya Emalı Sistemləri Konfransının materialları* (2017): 1729–1739.
[26] Priya Goyal və digərləru, "Dəqiq, Böyük Mini-Dəst SGD: ImageNet-in 1 Saatda Təlimi", *arXiv* ilkin nəşri, arXiv:1706.02677 (2017).

Neural şəbəkə hiperparametrlərinin tənzimlənməsi ilə bağlı daha yaxşı tətbiqlər(practices) üçün Leslie Smith tərəfindən 2018-ci ildə dərc olunmuş[27] əla məqaləyə baxa bilərsiniz.

Bu, süni neyron şəbəkələr və onların Keras ilə tətbiqi ilə bağlı girişimizi yekunlaşdırır. Növbəti bir neçə fəsildə çox dərin şəbəkələrin təlimi üçün texnikaları müzakirə edəcəyik. Həmçinin TensorFlow-un aşağı səviyyəli API-dən istifadə edərək modelləri necə fərdiləşdirmək və Data API vasitəsilə məlumatları səmərəli şəkildə yükləmək və ön emal etmək (preprocess) yollarını araşdıracağıq. Bundan əlavə, digər məşhur neyron şəbəkə arxitekturalarına da baxacağıq: şəkil emalı üçün konvolyutsiyalı neyron şəbəkələr, ardıcıl verilənlər üçün təkrarlayıcı neyron şəbəkələr, təmsil öyrənməsi(representation learning) üçün autoenkoderlər və verilənləri ümumiləşdirmək və modeli yaratmaq üçün generativ rəqib şəbəkələr.

Tapşırıqlar

1. TensorFlow Playground, TensorFlow komandası tərəfindən hazırlanmış əlverişli neyron şəbəkə simulyatorudur. Bu tapşırıqda siz bir neçə ikili klassifikasiya modelini bir neçə kliklə öyrədəcəksiniz və modelin arxitekturası ilə hiperparametrlərini dəyişdirərək neyron şəbəkələrin necə işlədiyi və hiperparametrlərin nə iş gördüyü barədə intuisiyanız formalaşacaq. Vaxt ayırıb aşağıdakılara göz gəzdirin:

    a. Neyron şəbəkə tərəfindən öyrənilən nümunələr**.** Standart neyron şəbəkəni "Run" düyməsini (sol üst künc) klikləyərək öyrədin. Diqqət et ki, şəbəkə klassifikasiya tapşırığı üçün sürətlə yaxşı həll yolu tapır. Birinci gizli qatın neyronları sadə naxışları öyrənərkən, ikinci gizli qatın neyronları isə birinci qatın sadə naxışlarını daha mürəkkəb nümunələrə birləşdirir. Qısacası, qatların sayı artdıqca, öyrənilə biləcək naxışlar daha mürəkkəb olur.

    b. Aktivasiya funksiyaları: Tanh aktivasiya funksiyasını ReLU ilə əvəz edib şəbəkəni yenidən öyrədin. Görəcəksiniz ki, həll daha sürətlə tapılır, lakin sərhədlər bu dəfə xətti olur. Bu ReLU funksiyasının forması ilə izah olunur.

    c. Lokal minimum riskləri: Şəbəkə arxitekturasını yalnız bir gizli qat və üç neyronlu edin. Şəbəkəni bir neçə dəfə öyrədin (şəbəkə çəkilərini sıfırlamaq üçün "Play" düyməsinin yanındakı "Reset" düyməsini klikləyin). Təlim vaxtının çox dəyişkən olduğunu və bəzən şəbəkənin lokal minimumda qaldığını müşahidə edin.

    d. Neyron şəbəkəsi həddən artıq kiçik olduqda baş verənlər: Bir neyronu çıxarın, yalnız iki neyronu saxlayın. Görəcəksiniz ki, hətta bir neçə dəfə sınasanız belə neyron şəbəkə artıq yaxşı həll tapa bilmir, Modelin parametrləri azdır və təlim dəstini sistematik olaraq underfit(zəif öyrənmə) edir.

[27] Leslie N. Smith, "Neyron Şəbəkələrinin Hiperparametrlərinə Disiplinli Yanaşma: I hissə — Öyrənmə Dərəcəsi, Dəst Ölçüsü, İmpuls və Çəki Azalması", *arXiv* ilkin nəşri, arXiv:1803.09820 (2018).

e. Neyron şəbəkə kifayət qədər böyük olduqda baş verənlər: Neyronların sayını səkkizə təyin edin və şəbəkəni bir neçə dəfə öyrədin. Görəcəksiniz ki,o indi ardıcıl olaraq sürətlidir və heç bir zaman ilişmir. Bu neyron şəbəkə nəzəriyyəsində vacib bir tapıntını göstərir: böyük neyron şəbəkələr demək olar ki, heç vaxt lokal minimumda ilişmir və ilişsələr belə, bu lokal optimumlar qlobal optimuma çox yaxındır. Lakin onlar uzun platformalarda ilişə bilərlər.

f. Dərin şəbəkələrdə qradientlərin yox olması riski: Spiral datasetini seçin ("DATA" bölməsində sağ-alt dataset), şəbəkə arxitekturasını dörd gizli qat və hər qat üçün səkkiz neyron olaraq dəyişdirin. Görəcəksiniz ki, təlim çox uzun çəkir və tez-tez uzun müddət platformalarda ilişir. Eyni zamanda, ən yüksək qatdakı neyronlar (sağ tərəfdəki) aşağı qatdakı neyronlardan (sol tərəfdəki) daha sürətlə inkişaf edir. Bu problem "qradientlərin yox olması" problemi adlanır və çəkilərin yaxşı başladılması, daha yaxşı optimallaşdırıcılar (AdaGrad və ya Adam) və ya Batch Normalization kimi texnikalarla yüngülləşdirilə bilər (Fəsil 11-də müzakirə olunur).

g. Daha irəli getmək: Bir saatlıq vaxt ayırıb digər parametrlərlə oynayın və neyron şəbəkələrin nə etdiyi ilə bağlı intuisiyanızı inkişaf etdirin.

2. A $\oplus$ B (A $\oplus$ B, burada $\oplus$ XOR əməliyyatını təmsil edir) hesablamaq üçün orjinal süni neyronlardan istifadə edərək ANN çəkin(Şəkil 10-3 dəki kimi). İpucu:$A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$

3. Nə üçün adətən klassik Perceptron (tək qatlı, hədd(threshold) məntiq vahidləri ilə öyrədilmiş) əvəzinə Logistik Regressiya təsnifatçısından istifadə etmək üstünlük təşkil edir? Perceptronu Logistik Regressiya təsnifatçısına ekvivalent etmək üçün necə tənzimləyə bilərsiniz?

4. Nə üçün logistik aktivasiya funksiyası ilk MLP-lərin öyrədilməsində əsas rol oynadı?

5. Üç məşhur aktivasiya funksiyasının adını sadalayın. Onları çəkə bilirsinizmi?

6. 10 ötürülmüş neyronlu bir input qatı , 50 süni neyronlu bir gizli qat , 3 süni neyronlu bir output qatından ibarət bir MLP təsəvvür edin. Bütün neyronlar ReLU aktivasiya funksiyasından istifadə edir.

- X input matrisasının forması nədir?
- Gizli qatın çəki (W_h) və sürüşmə (bias) (b_h) vektorlarının forması nədir?
- Output qatının çəki (W_o) və sürüşmə (bias) (b_o) vektorlarının forması nədir?
- Şəbəkənin output matrisası Y-nin forması nədir?
- Şəbəkənin output matrisasını Y, X, W_h, b_h, W_o və b_o funksiyası kimi hesablayan tənliyi yazın.

7. Emaili spam və ya ham olaraq təsnif etmək üçün sizə output qatında neçə neyron lazımdır? Output qatında hansı aktivasiya funksiyasından istifadə edərdiniz? Əgər bunun əvəzinə MNIST

verilənlər toplusu üzərində işləmək istəyirsinizsə, output qatında neçə neyrona ehtiyacınız olacaq və hansı aktivləşdirmə funksiyasından istifadə etməlisiniz? Bəs Fəsil 2-dəki kimi mənzil qiymətlərini proqnozlaşdırmaq üçün nə etmək lazımdır?

8. Geri yayılma (backpropagation) nədir və o, necə işləyir? Geri yayılma ilə tərs rejimli avtomatik diferensiallaşdırma (reverse-mode autodiff) arasında fərq nədir?

9. Sadə bir MLP-də dəyişdirə biləcəyiniz bütün hiperparametrləri sadalaya bilərsinizmi? Əgər MLP təlim verilənlərini həddindən artıq öyrənnərsə (overfitting), bu problemi aradan qaldırmaq üçün bu hiperparametrləri necə tənzimləyə bilərsiniz?

10. MNIST verilənlər toplusu üzərində dərin bir MLP modeli öyrədin (onu *keras.datasets.mnist.load_data()* funksiyası ilə yükləyə bilərsiniz). 98%-dən yuxarı dəqiqlik əldə edə bilib-bilməyəcəyinizi yoxlayın. Bu fəsildə təqdim olunan yanaşmadan istifadə edərək optimal öyrənmə dərəcəsini tapmağa çalışın (yəni öyrənmə dərəcəsini eksponent şəkildə artırın, itkini (loss) qrafikdə göstərin və itkinin kəskin artdığı nöqtəni müəyyən edin).Bütün əlavə üsulları da sınayın — saxlama nöqtələrini (checkpoints) yadda saxlayın, erkən dayandırma (early stopping) tətbiq edin və TensorBoard vasitəsilə öyrənmə əyrilərini çəkin.

Bu tapşırıqların həlləri Əlavə A-da təqdim olunub.