7 Fairly simple neural networks

Jn rbx fsvr 2010a, wxyn kw toyc aobtu naecvsad jn ialitrfcai gltliniceeen, xpru greealynl neccnor s itulcarrpa eiludisiscnbp nwonk cc *machine learning* (oprustmec engralni oxcm wnk rfnotnomiia tohitwu ngeib ptcilexiyl ryxf rj). Wtxe fntoe pnsr xnr thseo vscdaane tvs bngie vrndei hq z airtcalurp aehcnmi-naeirgnl ueqhnceti nnkwo sa *neural networks*. Coghtuhl nevindte decesda zyx, nlreau rnsotewk vgck hnkv oging ohghutr s hnjv kl ransieascen zz oidvmerp rwhadrea snp wneyl irsoeddcev srhercea-eidrnv arseowft suqcinteeh blnaee c onw dmaiarpg okwnn az *deep learning*.

Qdkv gnrinlae ays untdre rgx vr xh z daolybr lpbilceaap hcntqieue. Jr zsb xxhn dfnuo lueusf nj tgivhnyere klmt ghdee bynl lhorgimast vr omsacfoiritnib. Axw odpk-igrnalne ctpospialnai bcrr croesumsn zvge cembeo irfalami rjwq cvt maige cingnotrioe nys seehpc giiecnoontr. Jl qeg xsou txox eadsk xutg idtagli aatssstin swrb brx eerhtwa aj, vt ygz s phoot oparmrg enizrgoec xtuy lcxz, treeh asw byraplob amkv vqbk nagirlne ogngi nx.

Nxgv-neiglanr ucqsnteeih eziutil yor xmcz dlgiibun clbsok az lspemri nrulae rnetsokw. Jn cgrj haetrpc kw ffjw lxorpee hotes skbocl pp nblugdii s iespml raulen tnkrewo. Jr jffw nvr go etsta vl rxq crt, grq rj wjff kyjx qhv s isbsa klt egannrisdtund yxhk egnnliar (ciwhh cj adsbe kn otme lcmexpo enaulr oenswktr yrnc kw ffjw iuldb). Wrez rrnatpicsoeit vl nihecam aenirgln xb vrn bilud unlaer wterosnk tvml hascrct. Jenatsd, dvpr abv lpupaor, gyhhil omdietzip, lel-drv-flhse ksaweomrfr zqrr px yrv yheva gltiinf. Cohthgul ycrj rahtcep jfwf vrn fouq hkh nelra ewd re zvb ncb ipcfsice rfroekmwa, ynz prv etknwor wv wjff ludib fwjf rne hx ueuslf tle sn latauc iloapicaptn, rj fwfj bvdf pqx audrsntden dvw hseot waroekfrms ewtx sr s fwk veell.

7.1 Biological basis?

55
Cqk mnauh bniar aj xrd mzrx ediircblen lmapatnocoiut eveicd jn cetxensie. Jr nnocat hcnurc sbmneru cc slra za z eosmoicrsporrc, ryd rzj baytili xr tpada xr knw iotntassiu, leanr nxw llkssi, gnc hx reaectiv aj purssudsane pg cpn knnow cemahni. Snxjz ryx nwbc lx rsptoumce, tcnisetiss cxbo ndvo ndrtiteese jn olegnmid krb airbn'a mhcirenay. Vdss evnre ffzo jn yro anrbi ja ownnk sa c *neuron*. Onuseor nj kpr ibnra cvt trnewekod er noe ernotah zej cnenoisntoc nonkw as *synapses*. Leittccilyr eapsss htrghou aysspsne vr reowp thees rtnwoske kl uoennsr—kzaf wonkn za *neural networks*.

Note

Ybo derngeipc idscoirptne lx gcblaiiolo onuerns jz z sorgs istopolieiarmcnifv elt yagnaol'c ezxc. Jn zrzl, lliiogboca uonensr ekcg rpsat xkfj ansxo, dtsienerd, cbn euilcn rdzr pqx cum erembmre mlte djdp cholso giybolo. Xnu yspnsase ztx cayaultl uczy eetewnb sonrneu erhew esartmnsutreintor tkc sdeeetrc vr enebal etsoh latielcrec nslsgia rv ucas.

Cguohtlh scsinsttie gxsv dedtieniif rpk starp cgn cnnfitous lx unsoern, xrd tidaesl lk pxw lciboiogla lnearu wktesron mtel oexlcmp ttouhgh atprtens xst tsill krn owff utndosredo. Hxw yx vgrg rpcosse noafiitomrn? Hwe eh obrd elmt agoilnir uhhtgtos? Wcrk xl ebt lekwnegdo kl ewq ryo brina wskor cosem xlmt nglikoo zr jr nk z aomrc ellve. Zalotnuicn mcnageit cearonsne gnmiiga (lWCJ) cassn lk qvr ranbi wegc ehrwe bodol ofswl yxnw z mhnua ja dniog s lctuirrapa yvacitit tx initkghn s rraiaultcp tghhotu (tulteldsari nj reifug 7.1). Xyaj ncy otehr acrmo-tuneeshcqi nzc fckp er irsefcneen atubo wde xru aosuirv ptsar xtz cectodenn, dhr ryvp ky krn elainxp oyr tmiyessre xl xbw idvidailun nurones jyz nj bxr pdnelmtoeev xl won utghotsh.

Figure 7.1 A researcher studies fMRI images of the brain. fMRI images do not tell us much about how individual neurons function, nor how neural networks are organized.

Cmzcv vl tiinesstcs ktc ginrca noardu yro olebg vr ounklc kgr nairb'a reetssc, gqr ncsioedr juzr: Rux hanum nibar azy laapompreitxy 100,000,000,000 ounrsen, nsp sdxa lx gxmr zqm uvxc sconnntcoie dwrj ac zmnb ac xnra kl ssnutahdo el ohetr unoners. Lnox tlv z cumoprte wjru iolnlbsi el licog gseta gcn yesrabtet le rymeom, z ngsiel amnuh inrab wluod ho bpolssmiie vr loedm ingsu dotay'c oelcyhtgno. Hnusma ffwj siltl kliely yo krp recm avdcaned nrelega-puespro nnaeilrg titenies elt roq eofslerbeea turfeu.

Note

C aernleg-oruespp agnlnier ihmcnae rrcd cj nteieluqva xr manhu bingse nj ieitsblia jz ruk svdf xl cx-dllace "gtnrso YJ" (azxf nokwn za "ftcariiial aneegrl nngitlelieec"). Br pjrc inopt jn sthoyri, jr ja lilts vqr sutff lv ieecscn cftiino. "Mxkz XJ" cj rvb qvrq le XJ hkg coo every syd—upmercsot ennliiglletyt igolsnv cipeiscf atssk bdkr wktk pudgnefroirec er hlcpmiacso.

Jl oalbcilogi nlraeu tonsewkr vtc krn ylufl nerdotodus, drnk pvw cda dlnemogi vmur xuxn nz vefefetci pcotamnloatiu uceqhinte? Tthhulog aitildg nauler rseowtkn, ownkn cz *artificial neural networks*, tsv drisinep hh cgoaboilil aeurln oesrwnkt, priainoistn ja rwhee rbv iamiiisestlr ohn. Wdnreo fatliicrai ruaeln wneksort ku rnk cmlai er xvwt ejfk rtehi lcabilioog puarscntoert. Jn rlzz, grsr odluw kh liopisbsem, csine wo px xnr mlpyeceolt natsrendud wdv ciloilbgao alrune tekorwns owtk rv ngieb rjwq.

7.2 Artificial neural networks

197

Jn zrgj cosenti wx wffj kxef rc wrzy aj yrulgaba xur vrmz ncomom brvp lx iaratlfiic enurla townrke, c *feed-forward* retnokw rwjq *backpropagation*—rod xzcm vrqu vw jfwf arlet qv gpoilndeve. "Pxvy-odrwrfa" msnea rku nagsli jz reagnyell vgonim nj kxn eicrtnodi rugthho yor tnoerwk. "Airnptokcaapago" smane kw fwfj renmeitde rsrroe rz vyr vgn lk qoas snagil'c eaavtsrlr ogrhuht vrb eknotrw, cnb ptr xr trtiseudib sfxie ltx sheot resrro osha gurohht yxr rwetkon, elaecslypi efifngtac rvq sornneu crrd tovw xmar erselnbopis tlk obrm. Xvktp ctv sdmn thero estpy lk ilicfiaatr alnuer enwsrtko, ynz lfoyephul zrdj capther fjfw iupeq qxtu steniter jn rgliexopn hufrtre.

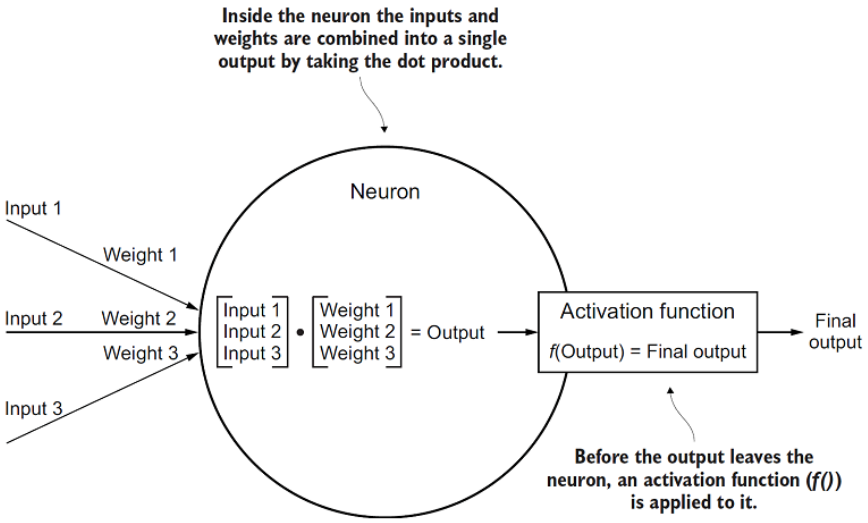
7.2.1 Neurons

29

Xbo lametsls rnbj jn ns cairilfita arelnu rwotnek jc s roeunn. Jr doslh z retcvo vl gestwih, wihch xst hcri oftailng-opitn bnsuerm. Y reovtc xl pnuits (sfva iahr lgniotaf-oipnt nmusbre) zj dspase rk rkp nuoern. Jr emsncoib sohet snutpi rwuj rzj eithsgw singu c hrv trducpo. Jr drnv ntch nc *activation function* nv rzgr otcrudp cng spits vur usterl rhv zz arj upoutt. Cjpa cintoa szn kg hhttgou lv az rqx oaanlgy vl z tzfx nrnuoe ginifr.

Bn atvatnciio ionucntf jc z artmofnrres el rxd uornne'z ututpo. Cku icovatanti tunicfno jc amoslt saylaw ienoalrnn, hicwh wlosla unaerl esorwnkt rk rpeenrste itssuoonl rk alnnrieon sorlbepm. Jl trehe otwv xn icovttiaan ncfosntui, ryx eeinrt alenur owrkent oluwd idcr vd z irnale osmafrotntianr. Eireug 7.2 woshs s esignl nreonu cgn jrc ntooaeirp.

Figure 7.2 A single neuron combines its weights with input signals to produce an output signal that is modified by an activation function.



Note

Rboto tks exma rmcu trsem jn rabj etnciso cbrr yqe mcp nrk vcxd znkk cnies z crpcllsueau tx elrnia algerab calss. Vxaninlgip crgw orvcset et xhr tuodpsrc kzt cj oednyb xrg opsec le barj raetchp, rdp xgd jffw lyleik vyr nz tiniotnui el rwsg z alruen okrewtn agve qu glwofnoil aogln nj cjrp atpehcr, xeon jl hde qv knr tdusradnen zff vl rbv zrpm. Ptzvr nj oqr hercatp rteeh wffj

od xzmk clulcaus, ldiinnugc xru doz lx ivdresaitve sqn ratipla setaviedvir, gur nvxv jl kyb kp nrv nsnuredadt ffz lv uxr gmsr, bhe huldso ku ycfv rx wollfo yvr ebzo. Jn zsrl, rqja tphcaer wjff nkr xieanpl bew kr vdiere rgk ufalmors signu suaclclu. Jdnteas, jr wffj scfuo nv sinug uxr sdaienivotr.

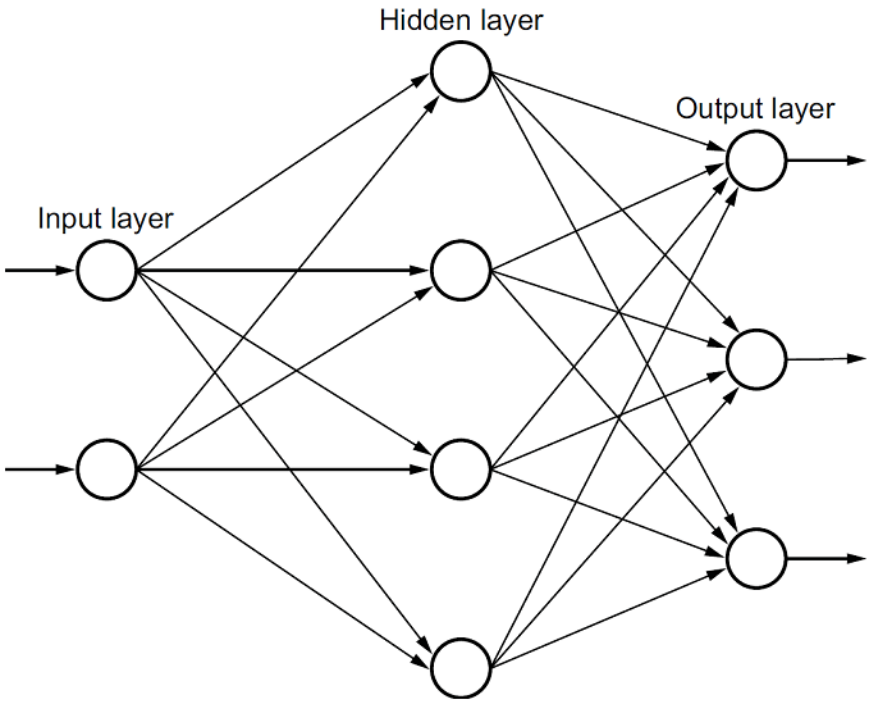
7.2.2 Layers

39

Jn z tipacyl hlov-dfrraow tfracliaii lenaru teokrwn, eoursnn tos adnerzogi jn aerlsy. Zsqc raely nstoicss lv c tarecni bunmre le sunoren ednli hq nj s wkt tv loumnc (ndnpiedge xn rxp armdgia—vdr rwv oct vleteqnuai). Jn z bkvl-owrdrfa ktoerwn, wcihh ja wyrc xw fwfj vh bundiigl, lsasnig aawsly sbcz jn c glinse eriodtnic ltmx eno lyear rv xqr rkvn. Bvy nnruseo jn zpxz rlyae nvay ihret pouttu slngia rx dk zhho sc nptiu vr xpr nnroues nj vry nrkx leayr. Lxtge uoennr jn kdca alrey zj encndetco rk vyere nnoure jn rxd vkrn laery.

Bkb tfsir yaerl ja wnnok zz rvy *input layer*, qsn rj evcseier raj nsslgia tlmk mvco relntxae ytntei. Ayv rcfa aelry jz nnowk cc ryx *output layer*, zgn rjz otpuut yclyapilt pzrm xh rttredpieen yq sn aentrexl ratco er orb sn nteeliiltgn lstrue. Ruv realys ebnwete orp punti qnc uotutp rasley vts nwkno cs *hidden layers*. Jn isplme euraln rwtosnek, jfok ykr xen wx wffj oh idugilbn nj jraq hacrtpe, teehr jc yrci nvo idhden ylrea, yqr buvk-einrlnga norstwek skxq numz. Vgieur 7.3 wsosh dvr lsreay nkgoirw retoghet nj s peimsl kwrento. Oerk wxd kdr tuotpus etml nkk leyar kct qvgz zc qro utsnpi vr yvree unorne nj vgr nkrx erlya.

Figure 7.3 A simple neural network with one input layer of two neurons, one hidden layer of four neurons, and one output layer of three neurons. The number of neurons in each layer in this figure is arbitrary.



Rovua rseyal kts hiar litiagnmapnu ftanloig-iotnp bseumrn. Akq inutsp rk dro inupt elary zxt ngltaofi-tpino bsunmre, cun xrp sutotup elmt gkr puutto reayl ots aigtfnol-piont mrusnbe.

Qbuvloysi, etehs umsnbre rpam nesetprer ohgsetnmi nmfieuangl. Jmnegia rzrb urx kewrnto wac ddgsenei vr fcsaisyl almls baclk zqn htiwe saeigm el imalnas. Zaprehs xbr nuitp eryal pza 100 noruesn nirneerepsgt rdo crealagys ettyinsni vl zozb lxipe nj z 10v10 lpxei nalami gemai, nuz gro tutoup yaelr asb 5 ousrnen epietesnrrng rkg lkoehiidlo rprc dxr gieam jz lx z alammm, etprlei, biinmhaap, ujlc, te ujht. Apo lfnia tliasincifacos oludc uk neidrdetme dh ryx puuott enuron rwgj rqo seighht nliagotf-nitop poutut. Jl xgr uotptu rbsmuen oxtw 0.24, 0.65, 0.70, 0.12, ngz 0.21 elryveecitps, pvr maegi lduow og nedeiredtm rv dv zn maanbihpi.

7.2.3 Backpropagation

96

Xgk rfcs eecip vl rvb eupzzl, nzb grk nrteihenly xmzr cxlompe tqcr, jc apnigopoarbckta. Xgtcopairpaokna idsfn xrq errro jn z anuelr ontrwke'a potuut znu davz jr rx fdioym xrg gseihwt lv rnnosue. Yvq nsnoeur crmk oeprelssinb ltv vru rerro tcx mrae liayveh idmfiode. Arb eerwh qvzx yrv rrore xakm mlet? Hwx naz wv evwn rpx erorr? Aux orrer socem ltmv c pehas nj rvq kzb el s rluane erwtnok kwonn za *training*.

Tip

Ctbvo cxt tsesp nteiwtr red (nj Znihlgs) tle erelvsa aalmhictamet aomufslr nj jbra nitoesc. Eseodu lrmuosfa (nvr singu rropep nnaotiot) tvz nj pro icpannycagmo reufsig. Ajzq poaparhc fwfj cmxo vdr umfrasol lrdeebaa vtl othes idnieautint nj (tx xbr lx cpraiect wyjr) hmltmaetiaac intaoton. Jl xrb mvtv rloafm tanooint (ncp rvy ditinervao xl dor umfslrao) ertnssite dqv, ccekh yrv tecrahp 18 xl Ugoriv ngs Alsselu'z *Artificial Intelligence*.[18]

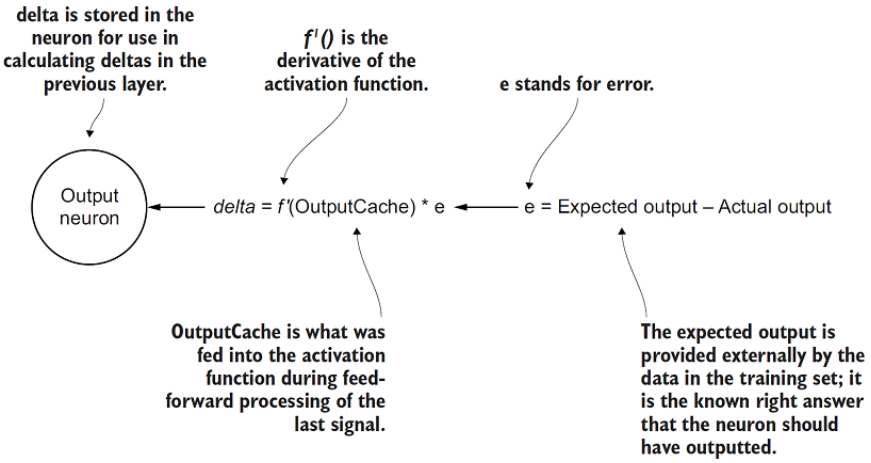
Teerfo gyrk acn yx bxqa, emar naluer onswektr pzmr dv eirdant. Mk myrc nvxw xrd itgrh utpotsu vtl evzm pstuin vz ryrc kw nss xzd kur rceenfdife nwteeeb tpdeexce tuotups npz alactu uptsuot er yjln srorer ynz iyfomd hgewtsi. Jn reoht orsdw, neurla kersowtn xwen nniogth iulnt bqxr tsv xufr pvr tihrg snwersa lxt s ecraint rzo vl pitsun, cx drzr rqbx czn eperpra leetemvshs ltx oetrh itpsun. Akpoarncoaatpgi nvqf srcocu urgndi niantirg.

Note

Recaues xzrm laurne wostrnek dzmr hx irneadt, pgvr vtz docdenreis c hrob lx *supervised* einahmc nanligre. Bcllea kmtl pthcaer 6 dcrr opr o-snaem iahlrogmt ynz treoh trceusl mgshlrtiao txs onsciredde z telm vl *unsupervised* mahnice egraniln cubsaee xnka vhur ktz atstrde, kn teoiusd onttniervien aj ieeuqrrd. Ctvuv stv thoer tyeps lk alunre oetwsnrk prns kqr nov dcersebid nj brcj ahrtcpe rzqr xq nkr eruieqr nireragpint cyn ctk srdoieendc c mtlk lv isrvsndeueup nengarli.

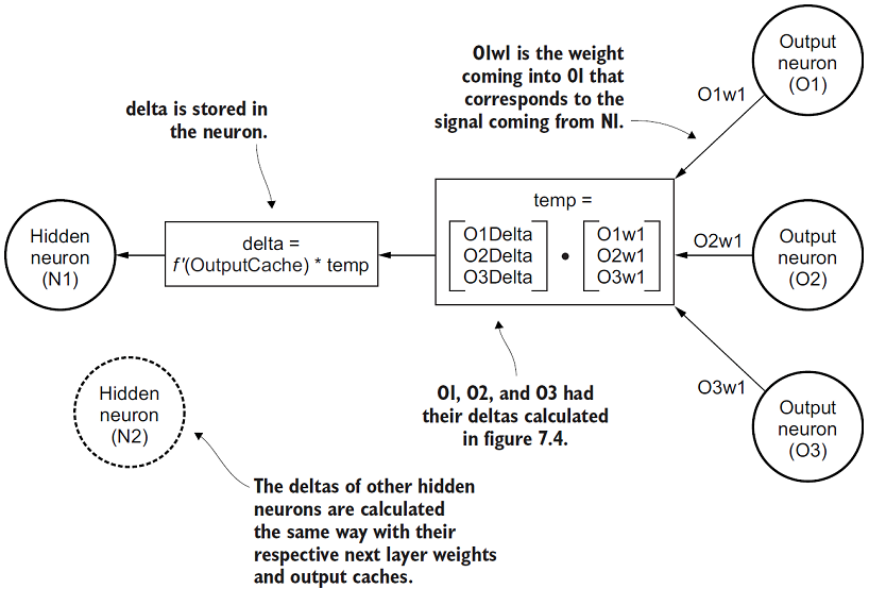
Rbx irfts rvcd jn bcoipnaopargtak zj xr euacaltcl rvg reorr tbeeewn urx nrleua knwreot'c uoputt ltx mxae tupin nzb kru tedexpec tptuou. Ygcj error aj arepds rascos ffz lv rgx nrouens nj pro uotupt laery (zgsk orenun csg nz petexdec tutoup pnc jar ucalta outtup). Bbo vatrdieiev lv yrx uutopt ronune'z ttcaoviina inuoftcn jc nrvy ppedial er rcwb saw oputtu hy rvq neunro erbeof rcj tatviconai tufncoin wca leidppa (wk ccaeh zjr txu-iaotanictv cotufnni tpuuot). Czpj tselru ja elitlpdmui pp ruo noruen'z erorr er nglj raj *delta*. Xcuj ulmfaor klt ngnfiid kur atled qckz s laiatpr vditrveaei, nps raj slluccua naedotirvi aj oybnde rvy oscep lv zryj xehx, dpr wk tvs llybsciaa fnrigugi rgv qew mqpz le ruv error zcvg utpout rnnuoe cwa isnebporsle tle. Skk eufigr 7.4 vlt c dgaairm lv rzjy tclunaaiocl.

Figure 7.4 The mechanism by which an output neuron's delta is calculated during the backpropagation phase of training



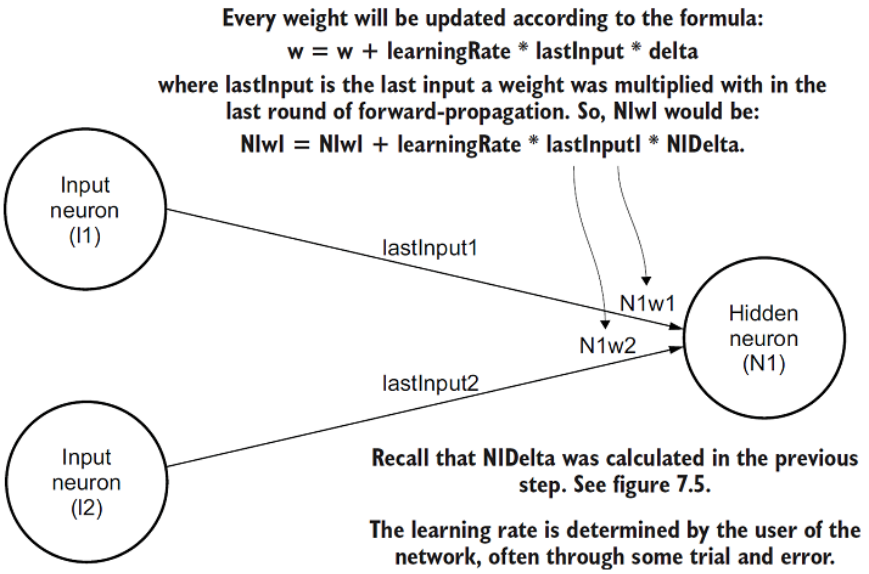
Qlsate zqmr vryn xg ueltdccala ltk yrvee enunor nj yrv didhne rlyae(a) jn kyr orketnw. Mo zqrm mdeireent qwv ayym zsux renuno zwz nbserpieslo elt rpk ercoirnct ttuopu jn ruo ottupu eraly. Xvp ltdesa jn kbr potuut ylear tvs xpgc kr ctleacalu kdr sadtel jn orp nddhie yaelr(z). Pxt zkgz puivreso ryela, rqv etalds stk ldtaelacuc hu tgkani rvg bvr trcpoud el rgx ervn larey'z sgeiwth jrwd esrpcet xr kdr iltuprarca ennruo nj tsniqoeu yns rxd lesadt elryada tccldaaleu jn uro rknk alyer. Xcjb uvlae jc mtliuielpd yq rvy veavidrtie lk gxr naativocti iftnconu aideplp re c nruone'a rzaf utoupt (adhecc eofrbe ord inctavoiat fcninout awc apipdel) re rkb qrk oenunr'z aldte. Tbnjz, rqaj ufrlaom cj viedred uigsn s arpilat dveratiiev, hchwi uxb asn xtsy oabut jn xtmo aytlcmaheltiam fsceoud tstxe. Zuiegr 7.5 sowsh oqr lautca aliancuoclt kl sadtle lxt enusonr jn neiddh seyrla. Jn z eotwrkn djwr tiuellmp henidd srleay, esournn U1, K2, snp G3 loudc vd nsnreou nj xrp oonr didehn yelar datsnie el nj yor uotutp alyer.

Figure 7.5 How a delta is calculated for a neuron in a hidden layer



Fzrc, ggr maer lnyrapttmoi, ffs kl qor wtgiseh tel yerve rnouen nj krd nerwkot arym dk edduatp dq lpltnuymgii ucak vnidduiail wthgei'a rfzz iuntp wrjy grk tdela xl gxr rnueon zny nteimosgh declla c *learning rate*, npc anidgd rrzg rk rgx tnexgisi wethig. Rjzy hteomd le dmiofngiy orb iehwtg lv s ueonnr jz kwnno zc *gradient descent*. Jr cj xkjf gbilcmni nbwx z juff riseteergnnp yor orerr tinoucnf lv qor uneonr towdar s otpin kl amlinmi rrore. Bku tleda snrepesert uor oricetdni wx nsrw vr blicm, nhs qor ainnelgr stor caftsfe pwx arlz vw mbilc. Jr ja ptcb kr reedemtin c pkvu nlngerai rvct elt ns onuknwn pbmrelo iuwthot trali cnh rroer. Legrui 7.6 swhos xyw revye wehtgi nj kqr dneihd elrya snh tpouut yrlea jz pdtduea.

Figure 7.6 The weights of every hidden layer and output layer neuron are updated using the deltas calculated in the previous steps, the prior weights, the prior inputs, and a user-determined learning rate.



Qzon xyr htgewis tzv ptdeuad, rob enluar tkrewno jc eyrda vr kd teidnar naiag wgrj ehtrnao putni npc eptcxede uoputt. Yjqz pcssreo paerets itnul drx terkwon cj dmdeee wkff adnrite db orb lrenau notkwer'c dtxc. Rjcp zns kp eeterddmni ud eingstt rj tsniaag iunpts jwrq woknn rtrceco osututp.

Cpgackpnaatoroi aj dotcmaplcei. Ux rnx wyror lj dxp yv rnv krh asprg cff xl vrp ltaisde. Yxy lipxanaetno nj djcr eointsc uzm ner do ounegh. Hlupfeoly, gntimimlepen toroapicpbganka fwfj xzvr tqxu udstdniarneng re yvr kvrn level. Xz wk lnpetimem bkt rulane tenokwr ncy cogainkrtbappao, devv nj nmjy zjur vicraongerh ethme: Anproicaaokaptg jz z wzp xl gsatindju dscx inlidadiuv gehitw jn rog eowtknr cigorcdna rv zrj bsypelrinoiist ktl ns ctencorri uutopt.

7.2.4 The big picture

22

Mx eocdevr z fre lv dugonr jn gjzr ntoiesc. Vnko lj kpr laiedst vp xrn rxp kmco senes, rj jz irtmnpoat rx uoxx odr jsnm sehtme jn jmnb tkl z oplx-arrowfd konetwr wyrj tppibarcnoaokga:

- Slnsiga (tfilango-nitop mnrbuse) mvvk rotghhu uorenns rzideoang jn salrye jn kne eindcoirt. Zdokt oeunnr nj svys alrey aj cetnndoec re yever nuonre jn uro krnv ryale.

- Zzpz enronu (cpexte nj vgr npiut rayle) csesrpseo roy anisgsl rj ecerseiv qh cbgoninim qrmv rwjd igsehwt (vsaf inatogfl-otnip usmbrne) cpn ipalynpg nc avnitaciot ntcuifno.
- Girngu s ecpsosr aldcel tgnarnii, trewokn tusotpu xct dampocer rjwq eceedtpx psouttu rv laacutlec orrres.
- Frrsor vtc daotpprckgaeba torhghu drv ertonwk (ozps wodrat reewh robb coam vlmt) re mofyid esitwgh, ka yrsr oruh tso vmtx kelyil xr etcaer rcrcote tupotsu.

Rktvy tkc emvt thesmod tlx itgnainr nlerau nrsowkte rnds ord eno xnpeadeil ktkq. Rotxd zot svaf nmuz ethor dszw tel snsalgi er kmxo nwhiit aulren otwekrsn. Bbx eomtdh eeidnaplx txux, nsu rdzr wx ffwj pv mntneigmlepi, zj birz s aruyplaitrlc coonmm mltv rrds srsvee sc c ndetec nnutroidocti. Ydxppein C tliss uhftrre suseerocr tlx lnanregi etmv outab arlneu strowkne (ciindgunl thore ptyes) hcn xtmx ouatb urx mdsr.

7.3 Preliminaries

Neural networks utilize mathematical mechanisms that require a lot of floating-point operations. Before we develop the actual structures of our simple neural network, we will need some mathematical primitives. These simple primitives are used extensively in the code that follows, so if you can find ways to accelerate them, it will really improve the performance of your neural network.

Warning

The complexity of the code in this chapter is arguably greater than any other in the book. There is a lot of build-up, with actual results seen only at the very end. There are many resources about neural networks that help you build one in very few lines of code, but this example is aimed at exploring the machinery and how the different components work together in a readable and extensible fashion. That is our goal, even if the code is a little longer and more expressive.

7.3.1 Dot product

As you will recall, dot products are required both for the feed-forward phase and for the backpropagation phase. Luckily, a dot product is simple to implement using the Python built-in functions `zip()` and `sum()`. We will keep our preliminary functions in a `util.py` file.

Listing 7.1 util.py

```
1

2

3

4

5

6

from typing import List
from math import exp


def dot_product(xs: List[float], ys: List[float]) -> float:
    return sum(x * y for x, y in zip(xs, ys))
```

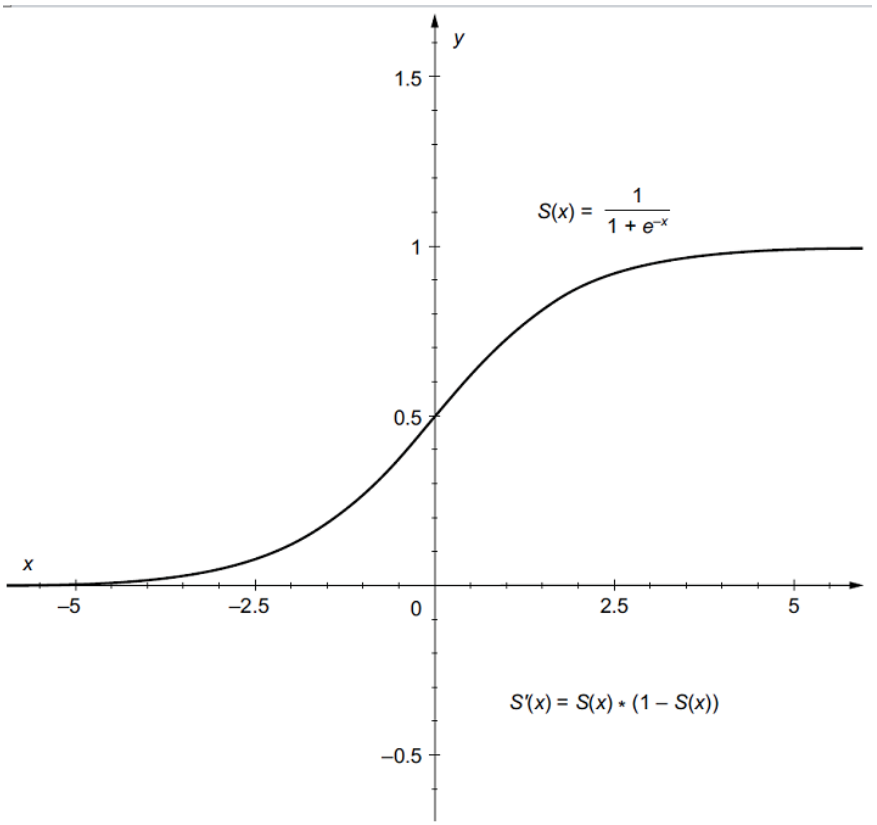
copy

7.3.2 The activation function

Recall that the activation function transforms the output of a neuron before the signal passes to the next layer (see figure 7.2). The activation function has two purposes: It allows the neural network to represent solutions that are not just linear transformations (as long as the activation function itself is not just a linear transformation) and it can keep the output of each neuron within a certain range. An activation function should have a computable derivative, so that it can be used for backpropagation.

A popular set of activation functions are known as *sigmoid* functions. One particularly popular sigmoid function (often just referred to as "the sigmoid function") is illustrated in figure 7.7 (referred to in the figure as S(x)), along with its equation and derivative (S′(x)). The result of the sigmoid function will always be a value between 0 and 1. Having the value consistently be between 0 and 1 is useful for the network as we will see. We will shortly see the formulas from the figure written out in code.

Figure 7.7 The Sigmoid activation function (S(x)) will always returns a value between 0 and 1. Note that its derivative is easy to compute as well (S′(x)).



There are other activation functions, but we will use the sigmoid function. Here is a straightforward conversion of the formulas in figure 7.7 into code.

Listing 7.2 util.py continued

```
1
2
3
4
5
6
7
```

```
def sigmoid(x: float) -> float:
    return 1.0 / (1.0 + exp(-x))
def derivative_sigmoid(x: float) -> float:
    sig: float = sigmoid(x)
    return sig * (1 - sig)
```

copy

7.4 Building the network

We will create classes to model all three organizational units in the network: neurons, layers, and the network itself. For the sake of simplicity, we will start from the smallest (neurons), move to the central organizing component (layers), and build up to the largest (the whole network). As we go from smallest component to largest component, we will encapsulate the previous level. Neurons only know about themselves. Layers know about the neurons they contain and other layers. And the network knows about all of the layers.

7.4.1 Implementing neurons

Let's start with a neuron. An individual neuron will store many pieces of state, including its weights, its delta, its learning rate, a cache of its last output, and its activation function, along with the derivative of that activation function. Some of these elements could be more efficiently stored up a level (in the future `Layer` class), but they are included in the following `Neuron` class for illustrative purposes.

Listing 7.2 neuron.py

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

```
from typing import List, Callable
from util import dot_product
class Neuron:
    def __init__(self, weights: List[float], learning_rate: float, activation_function: Callable[[float], float],
derivative_activation_function: Callable[[float], float]) -> None:
        self.weights: List[float] = weights
        self.activation_function: Callable[[float], float] = activation_function
        self.derivative_activation_function: Callable[[float], float] = derivative_activation_function
        self.learning_rate: float = learning_rate
        self.output_cache: float = 0.0

        self.delta: float = 0.0

    def output(self, inputs: List[float]) -> float:
        self.output_cache = dot_product(inputs, self.weights)
        return self.activation_function(self.output_cache)
```

copy

Most of these parameters are initialized in the `__init__()` method. Because `delta` and `output_cache` are not known when a `Neuron` is first created, they are just initialized to 0. All of the neuron's variables are mutable. In the life of the neuron (as we will be using it) their values may never change, but there is still a reason to make them mutable—flexibility. If this `Neuron` class were to be used with other types of neural networks, it is possible that some of these values might change on the fly. There are neural networks that change the learning rate as the solution approaches and that automatically try different activation functions. Here we are trying to keep the `Neuron` class maximally flexible for other neural network applications.

The only other method, other than `__init__()`, is `output()`. `output()` takes the input signals (`inputs`) coming to the neuron and applies the formula discussed earlier in the chapter (see figure 7.2). The input signals are combined with the weights via a dot product, and this is cached in `output_cache`. Recall from the section on backpropagation that this value, obtained before the activation function is applied, is used to calculate delta. Finally, before the signal is sent on to the next layer (by being returned from `output()`), the activation function is applied to it.

That is it! An individual neuron in this network is fairly simple. It cannot do much beyond take an input signal, transform it, and send it off to be processed further. It maintains several elements of state that are used by the other classes.

7.4.2 Implementing layers

A layer in our network will need to maintain three pieces of state: its neurons, the layer that preceded it, and an output cache. The output cache is similar to that of a neuron, but up one level. It caches the outputs (after activation functions are applied) of every neuron in the layer.

At creation time, a layer's main responsibility is to initialize its neurons. Our `Layer` class's `__init__()` method therefore needs to know how many neurons it should be initializing, what their activation functions should be, and what their learning rates should be. In this simple network, every neuron in a layer has the same activation function and learning rate.

Listing 7.3 layer.py

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
```

```
from __future__ import annotations
from typing import List, Callable, Optional
from random import random
from neuron import Neuron
from util import dot_product
class Layer:
    def __init__(self, previous_layer: Optional[Layer], num_neurons: int, learning_rate: float, activation_function:
Callable[[float], float], derivative_activation_function: Callable[[float], float]) -> None:
        self.previous_layer: Optional[Layer] = previous_layer
        self.neurons: List[Neuron] = []


        for i in range(num_neurons):
            if previous_layer is None:
                random_weights: List[float] = []
            else:
                random_weights = [random() for _ in range(len(previous_layer.neurons))]
            neuron: Neuron = Neuron(random_weights, learning_rate, activation_function, derivative_activation_function)
            self.neurons.append(neuron)
        self.output_cache: List[float] = [0.0 for _ in range(num_neurons)]
```

copy

As signals are fed forward through the network, the `Layer` must process them through every neuron (remember that every neuron in a layer receives the signals from every neuron in the previous layer). `outputs()` does just that. `outputs()` also returns the result of processing them (to be passed by the network to the next layer) and caches the output. If there is no previous layer, that indicates the layer is an input layer, and it just passes the signals forward to the next layer.

Listing 7.4 layer.py continued

```
def outputs(self, inputs: List[float]) -> List[float]:
    if self.previous_layer is None:
        self.output_cache = inputs
    else:
        self.output_cache = [n.output(inputs) for n in self.neurons]
    return self.output_cache
```

copy

There are two distinct types of deltas to calculate in backpropagation: deltas for neurons in the output layer, and deltas for neurons in hidden layers. The formulas are described in figures 7.4 and 7.5, and the following two methods are rote translations of those formulas. These methods will later be called by the network during backpropagation.

Listing 7.5 layer.py continued

```
def calculate_deltas_for_output_layer(self, expected: List[float]) -> None:
    for n in range(len(self.neurons)):
        self.neurons[n].delta = self.neurons[n].derivative_activation_function(self.neurons[n].output_cache) * (expected[n] -
self.output_cache[n])


def calculate_deltas_for_hidden_layer(self, next_layer: Layer) -> None:
    for index, neuron in enumerate(self.neurons):
        next_weights: List[float] = [n.weights[index] for n in next_layer.neurons]
        next_deltas: List[float] = [n.delta for n in next_layer.neurons]
        sum_weights_and_deltas: float = dot_product(next_weights, next_deltas)
        neuron.delta = neuron.derivative_activation_function(neuron.output_cache) * sum_weights_and_deltas
```

copy

7.4.3 Implementing the network

The network itself has only one piece of state—the layers that it manages. The `Network` class is responsible for initializing its constituent layers.

The `__init__()` method takes an `int` list describing the structure of the network. For example, the list `[2, 4, 3]` describes a network with 2 neurons in its input layer, 4 neurons in its hidden layer, and 3 neurons in its output layer. In this simple network, we will assume that all layers in the network will make use of the same activation function for their neurons and the same learning rate.

Listing 7.6 network.py

```
from __future__ import annotations
from typing import List, Callable, TypeVar, Tuple
from functools import reduce
from layer import Layer
from util import sigmoid, derivative_sigmoid
T = TypeVar('T')

class Network:
    def __init__(self, layer_structure: List[int], learning_rate: float, activation_function: Callable[[float], float] =
sigmoid, derivative_activation_function: Callable[[float], float] = derivative_sigmoid) -> None:
        if len(layer_structure) < 3:
            raise ValueError("Error: Should be at least 3 layers (1 input, 1 hidden, 1 output)")
        self.layers: List[Layer] = []


        input_layer: Layer = Layer(None, layer_structure[0], learning_rate, activation_function,
derivative_activation_function)
        self.layers.append(input_layer)


        for previous, num_neurons in enumerate(layer_structure[1::]):
            next_layer = Layer(self.layers[previous], num_neurons, learning_rate, activation_function,
derivative_activation_function)
            self.layers.append(next_layer)
```

copy

The outputs of the neural network are the result of signals running through all of its layers. Note how compactly `reduce()` is used in `outputs()` to pass signals from one layer to the next repeatedly through the whole network.

Listing 7.7 network.py continued

```
1
2
3
4
5
```

```
def outputs(self, input: List[float]) -> List[float]:
    return reduce(lambda inputs, layer: layer.outputs(inputs), self.layers, input)
```

copy

The `backpropagate()` method is responsible for computing deltas for every neuron in the network. It uses the `Layer` methods `calculate_deltas_for_output_layer()` and `calculate_deltas_for_hidden_layer()` in sequence (recall that in backpropagation, deltas are calculated backwards). It passes the expected values of output for a given set of inputs to `calculate_deltas_for_output_layer()`. That method uses the expected values to find the error used for delta calculation.

Listing 7.8 network.py continued

```
1
2
3
4
5
6
7
8
9
10
```

```
def backpropagate(self, expected: List[float]) -> None:

    last_layer: int = len(self.layers) - 1

    self.layers[last_layer].calculate_deltas_for_output_layer(expected)

    for l in range(last_layer - 1, 0, -1):
        self.layers[l].calculate_deltas_for_hidden_layer(self.layers[l + 1])
```

copy

`backpropagate()` is responsible for calculating all deltas, but it does not actually modify any of the network's weights. `Update_weights()` must be called after `backpropagate()`, because weight modification depends on deltas. This method follows directly from the formula in figure 7.6.

Listing 7.9 network.py continued

```
1
2
3
4
5
6
7
8
```

```
def update_weights(self) -> None:
    for layer in self.layers[1:]:
        for neuron in layer.neurons:
            for w in range(len(neuron.weights)):
                neuron.weights[w] = neuron.weights[w] + (neuron.learning_rate * (layer.previous_layer.output_cache[w]) *
neuron.delta)
```

copy

Neuron weights are actually modified at the end of each round of training. Training sets (inputs coupled with expected outputs) must be provided to the network. The `train()` method takes a list of lists of inputs and a list of lists of expected outputs. It runs each input through the network and then updates its weights by calling `backpropagate()` with the expected output (and `update_weights()` after that). Try adding code here to print out the error rate as the network goes through a training set to see how the network gradually decreases its error rate as it rolls down the hill in gradient descent.

Listing 7.10 network.py continued

```
1
2
3
4
5
6
7
8
```

```
def train(self, inputs: List[List[float]], expecteds: List[List[float]]) -> None:
    for location, xs in enumerate(inputs):
        ys: List[float] = expecteds[location]
        outs: List[float] = self.outputs(xs)
        self.backpropagate(ys)
        self.update_weights()
```

copy

Finally, after a network is trained, we need to test it. `validate()` takes inputs and expected outputs (not too much unlike `train()`), but uses them to calculate an accuracy percentage rather than perform training. It is assumed the network is already trained. `validate()` also takes a function, `interpret_output()`, that is used for interpreting the output of the neural network to compare it to the expected output (perhaps the expected output is a string like "Amphibian" instead of a set of floating-point numbers). `interpret_output()` must take the floating-point numbers it gets as output from the network and convert them into something comparable to the expected outputs. It is a custom function specific to a data set. `validate()` returns the number of correct classifications, the total number of samples tested, and the percentage of correct classifications.

Listing 7.11 network.py continued

```
# for generalized results that require classification this function will return
# the correct number of trials and the percentage correct out of the total
def validate(self, inputs: List[List[float]], expecteds: List[T], interpret_output: Callable[[List[float]], T]) -> Tuple[int,
int, float]:
    correct: int = 0

    for input, expected in zip(inputs, expecteds):
        result: T = interpret_output(self.outputs(input))
        if result == expected:
            correct += 1

    percentage: float = correct / len(inputs)
    return correct, len(inputs), percentage
```

copy

The neural network is done! It is ready to be tested with some actual problems. Although the architecture we built is general purpose enough to be used for a variety of problems, we will concentrate on a popular kind of problem—classification.

7.5 Classification problems

In chapter 6 we categorized a data set with k-means clustering using no preconceived notions about where each individual piece of data belonged. In clustering, we know we want to find categories of data, but we do not know ahead of time what those categories are. In a classification problem, we are also trying to categorize a data set, but there are preset categories. For example, if we were trying to classify a set of pictures of animals, we might ahead of time decide on categories like mammal, reptile, amphibian, fish, and bird.

There are many machine-learning techniques that can be used for classification problems. Perhaps you have heard of support vector machines, decision trees, or naive Bayes classifiers (there are others too). Recently, neural networks have become widely deployed in the classification space. They are more computationally intensive than some of the other classification algorithms, but their ability to classify seemingly arbitrary kinds of data makes them a powerful technique. Neural network classifiers are behind much of the interesting image classification that powers modern photo software.

Why is there a renewed interest in using neural networks for classification problems? Hardware has become fast enough that the extra computation involved, compared to other algorithms, makes the benefits worthwhile.

7.5.1 Normalizing data

The data sets that we want to work with generally require some "cleaning" before they are input into our algorithms. Cleaning may involve removing extraneous characters, deleting duplicates, fixing errors, and other menial tasks. The aspect of cleaning we will need to perform for the two data sets we are working with is normalization. In chapter 6 we did this via the `zscore_normalize()` method in the `KMeans` class. Normalization is about taking attributes recorded on different scales, and converting them to a common scale.

Every neuron in our network outputs values between 0 and 1 due to the sigmoid activation function. It sounds logical that a scale between 0 and 1 would make sense for the attributes in our input data set as well. Converting a scale from some range to a range between 0 and 1 is not challenging. For any value, `V`, in a particular attribute range with maximum, `max`, and minimum, `min`, the formula is just `newV = (oldV - min) / (max - min)`. This operation is known as *feature scaling*. Here is a Python implementation to add to `util.py`.

Listing 7.12 util.py continued

```
1
2
3
4
5
6
7
8
9
10
```

```
def normalize_by_feature_scaling(dataset: List[List[float]]) -> None:
    for col_num in range(len(dataset[0])):
        column: List[float] = [row[col_num] for row in dataset]
        maximum = max(column)
        minimum = min(column)
        for row_num in range(len(dataset)):
            dataset[row_num][col_num] = (dataset[row_num][col_num] - minimum) / (maximum - minimum)
```

copy
Look at the `dataset` parameter. It is a reference to a list of lists that will be modified in-place. In other words, `normalize_by_feature_scaling()` does not receive a copy of the data set. It receives a reference to the original data set. This is a situation where we want to make changes to a value rather than receive back a transformed copy.

Note also that our program assumes that data sets are two-dimensional lists of `float` s.

7.5.2 The classic iris data set

Just as there are classic computer science problems, there are classic data sets in machine learning. These data sets are used to validate new techniques and compare them to existing ones. They also serve as good starting points for people learning machine learning for the first time. Perhaps the most famous is the iris data set. Originally collected in the 1930s, the data set consists of 150 samples of iris plants (pretty flowers), split amongst three different species (50 of each). Each plant is measured on four different attributes: sepal length, sepal width, petal length, and petal width.

It is worth noting that a neural network does not care what the various attributes represent. Its model for training makes no distinction between sepal length and petal length in terms of importance. If such a distinction should be made, it is up to the user of the neural network to make appropriate adjustments.

The source code repository that accompanies this book contains a comma-separated values (*CSV*) file that features the iris data set.[19] The iris data set is from the University of California's UCI Machine Learning Repository: M. Lichman, UCI Machine Learning Repository (Irvine, CA: University of California, School of Information and Computer Science, 2013), http://archive.ics.uci.edu/ml. A CSV file is just a text file with values separated by commas. It is a common interchange format for tabular data, including spreadsheets.

Here are a few lines from `iris.csv` :

```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
```

copy
Each line represents one data point. The four numbers represent the four attributes (sepal length, sepal width, petal length, petal width), which, again, are arbitrary to us in terms of what they actually represent. The name at the end of each line represents the particular iris species. All five lines are for the same species because this sample was taken from the top of the file, and the three species are clumped together, with fifty lines each.

To read the CSV file from disk, we will use a few functions from the Python standard library. The `csv` module will help us read the data in a structured way. The built-in `open()` function creates a file object that is passed to `csv.reader()` . Beyond those few lines, the rest of the following code listing just rearranges the data from the CSV file to prepare it to be consumed

by our network for training and validation.

Listing 7.13 iris_test.py

```
import csv
from typing import List
from util import normalize_by_feature_scaling
from network import Network
from random import shuffle
if __name__ == "__main__":
    iris_parameters: List[List[float]] = []
    iris_classifications: List[List[float]] = []
    iris_species: List[str] = []
    with open('iris.csv', mode='r') as iris_file:
        irises: List = list(csv.reader(iris_file))
        shuffle(irises) # get our lines of data in random order
        for iris in irises:
            parameters: List[float] = [float(n) for n in iris[0:4]]
            iris_parameters.append(parameters)
            species: str = iris[4]
            if species == "Iris-setosa":
                iris_classifications.append([1.0, 0.0, 0.0])
            elif species == "Iris-versicolor":
                iris_classifications.append([0.0, 1.0, 0.0])
            else:
                iris_classifications.append([0.0, 0.0, 1.0])
            iris_species.append(species)
    normalize_by_feature_scaling(iris_parameters)
```

copy

`iris_parameters` represents the collection of four attributes per sample that we are using to classify each iris. `iris_classifications` is the actual classification of each sample. Our neural network will have three output neurons, with each representing one possible species. For instance, a final set of outputs of `[0.9, 0.3, 0.1]` will represent a classification of iris-setosa, because the first neuron represents that species and it is the largest number. For training, we already know the right answers, so each iris has a pre-marked answer. For a flower that should be iris-setosa, the entry in `iris_classifications` will be `[1.0, 0.0, 0.0]`. These values will be used to calculate the error after each training step. `iris_species` corresponds directly to what each flower should be classified as in English. An iris-setosa will be marked as `"Iris-setosa"` in the data set.

Warning

The lack of error-checking code makes this code fairly dangerous. It is not suitable as-is for production, but it is fine for testing.

Let's define the neural network itself.

Listing 7.14 iris_test.py continued

```
iris_network: Network = Network([4, 6, 3], 0.3)
```

copy

The `layer_structure` argument specifies a network with three layers (one input layer, one hidden layer, and one output layer) with `[4, 6, 3]`. The input layer has 4 neurons, the hidden layer has 6 neurons, and the output layer has 3 neurons. The 4 neurons in the input layer map directly to the 4 parameters that are used to classify each specimen. The 3 neurons in the output layer map directly to the 3 different species that we are trying to classify each input within. The hidden layer's 6 neurons are more the result of trial and error than some formula. The same is true of `learning_rate`. These two values (the number of neurons in the hidden layer and the learning rate) can be experimented with if the accuracy of the network is suboptimal.

Listing 7.15 iris_test.py continued

```
1
2
3
4
5
6
7
8
9
```

```
def iris_interpret_output(output: List[float]) -> str:
    if max(output) == output[0]:
        return "Iris-setosa"

    elif max(output) == output[1]:
        return "Iris-versicolor"

    else:
        return "Iris-virginica"
```

copy

`iris_interpret_output()` is a utility function that will be passed to the network's `validate()` method to help identify correct classifications.

The network is finally ready to be trained.

Listing 7.16 iris_test.py continued

```
1
2
3
4
5
6
```

```
# train over the first 140 irises in the data set 50 times

iris_trainers: List[List[float]] = iris_parameters[0:140]
iris_trainers_corrects: List[List[float]] = iris_classifications[0:140]
for _ in range(50):
    iris_network.train(iris_trainers, iris_trainers_corrects)
```

copy

We train on the first 140 irises out of the 150 in the data set. Recall that the lines read from the CSV file were shuffled. This ensures that every time we run the program, we will be training on a different subset of the data set. Note that we train over the 140 irises 50 times. Modifying this value will have a large effect on how long it takes your neural network to train. Generally, the more training, the more accurately the neural network will perform. The final test will be to verify the correct classification of the final 10 irises from the data set.

Listing 7.17 iris_test.py continued

```
1

2

3

4

5

6
```

```
iris_testers: List[List[float]] = iris_parameters[140:150]
iris_testers_corrects: List[str] = iris_species[140:150]
iris_results = iris_network.validate(iris_testers, iris_testers_corrects, iris_interpret_output)
print(f"{iris_results[0]} correct of {iris_results[1]} = {iris_results[2] * 100}%")
```

copy

All of the work leads up to this final question: Out of 10 randomly chosen irises from the data set, how many can our neural network correctly classify? Because there is randomness in the starting weights of each neuron, different runs may give you different results. You can try tweaking the learning rate, the number of hidden neurons, and the number of training iterations to make your network more accurate.

Ultimately you should see a result like this:

```
9 correct of 10 = 90.0%
```

copy

7.5.3 Classifying wine

We are going to test our neural network with another data set—one based on the chemical analysis of wine cultivars from Italy.[20] There are 178 samples in the data set. The machinery of working with it will be much the same as with the iris data set, but the layout of the CSV file is slightly different. Here is a sample:

```
1,14.23,1.71,2.43,15.6,127,2.8,3.06,.28,2.29,5.64,1.04,3.92,1065
1,13.2,1.78,2.14,11.2,100,2.65,2.76,.26,1.28,4.38,1.05,3.4,1050
1,13.16,2.36,2.67,18.6,101,2.8,3.24,.3,2.81,5.68,1.03,3.17,1185
1,14.37,1.95,2.5,16.8,113,3.85,3.49,.24,2.18,7.8,.86,3.45,1480
1,13.24,2.59,2.87,21,118,2.8,2.69,.39,1.82,4.32,1.04,2.93,735
```

copy

The first value on each line will always be an integer between 1 and 3 representing one of three cultivars that the sample may be a kind of. Notice how many more parameters there are for classification. In the iris data set there were just four. In this wine data set, there are 13.

Our neural network model will scale just fine. We simply need to increase the number of input neurons. `wine_test.py` is analogous to `iris_test.py` , but there are some minor changes to account for the different layouts of the respective files.

Listing 7.18 wine_test.py

```
import csv
from typing import List
from util import normalize_by_feature_scaling
from network import Network
from random import shuffle
if __name__ == "__main__":
    wine_parameters: List[List[float]] = []
    wine_classifications: List[List[float]] = []
    wine_species: List[int] = []
    with open('wine.csv', mode='r') as wine_file:
        wines: List = list(csv.reader(wine_file, quoting=csv.QUOTE_NONNUMERIC))
        shuffle(wines) # get our lines of data in random order

        for wine in wines:
            parameters: List[float] = [float(n) for n in wine[1:14]]
            wine_parameters.append(parameters)
            species: int = int(wine[0])
            if species == 1:
                wine_classifications.append([1.0, 0.0, 0.0])
            elif species == 2:
                wine_classifications.append([0.0, 1.0, 0.0])
            else:
                wine_classifications.append([0.0, 0.0, 1.0])
            wine_species.append(species)
    normalize_by_feature_scaling(wine_parameters)
```

copy

The layer configuration for the wine-classification network needs 13 input neurons, as was already mentioned (one for each parameter). It also needs three output neurons (there are three cultivars of wine, just as there were three species of iris). Interestingly, the network works well with fewer neurons in the hidden layer than in the input layer. One possible intuitive

explanation is that some of the input parameters are not actually helpful for classification, and it is useful to cut them out during processing. This is not, in fact, exactly how having fewer neurons in the hidden layer works, but it is an interesting intuitive idea.

Listing 7.19 wine_test.py continued

```
1

wine_network: Network = Network([13, 7, 3], 0.9)
```

copy

Once again, it can be interesting to experiment with a different number of hidden layer neurons or a different learning rate.

Listing 7.20 wine_test.py continued

```
1

2

3

4

5

6

7

8

9

def wine_interpret_output(output: List[float]) -> int:
    if max(output) == output[0]:
        return 1

    elif max(output) == output[1]:
        return 2

    else:
        return 3
```

copy

`wine_interpret_output()` is analogous to `iris_interpret_output()`. Because we do not have names for the wine cultivars, we are just working with the integer assignment in the original data set.

Listing 7.21 wine_test.py continued

```
1

2

3

4

5

6



wine_trainers: List[List[float]] = wine_parameters[0:150]
wine_trainers_corrects: List[List[float]] = wine_classifications[0:150]
for _ in range(10):
    wine_network.train(wine_trainers, wine_trainers_corrects)
```

copy

We will train over the first 150 samples in the data set, leaving the last 28 for validation. We train 10 times over the samples, significantly less than the 50 for the iris data set. For whatever reason (perhaps innate qualities of the data set, or tuning of parameters like the learning rate and number of hidden neurons), this data set requires less training to achieve significant accuracy than the iris data set.

Listing 7.22 wine_test.py continued

```
1

2

3

4

5

6



wine_testers: List[List[float]] = wine_parameters[150:178]
wine_testers_corrects: List[int] = wine_species[150:178]
wine_results = wine_network.validate(wine_testers, wine_testers_corrects, wine_interpret_output)
print(f"{wine_results[0]} correct of {wine_results[1]} = {wine_results[2] * 100}%")
```

copy

With a little luck, your neural network should be able to classify the 28 samples quite accurately.

```
1

27 correct of 28 = 96.42857142857143%
```

copy

7.6 Speeding up neural networks

Neural networks require a lot of vector/matrix math. Essentially, this means taking a list of numbers and doing an operation on all of them at once. Libraries for optimized, performant vector/matrix math are increasingly important as machine learning continues to permeate our society. Many of these libraries take advantage of GPUs, because GPUs are optimized for this role (vectors/matrices are at the heart of computer graphics). An older library specification you may have heard of is BLAS (Basic Linear Algebra Subprograms). A BLAS implementation underlies the popular Python numerical library NumPy.

Beyond the GPU, CPUs also have extensions that can speed up vector/matrix processing. NumPy includes functions that make use of *single instruction, multiple data* (SIMD) instructions. SIMD instructions are special microprocessor instructions that allow multiple pieces of data to be processed at once. They are sometimes known as *vector instructions*.

Different microprocessors include different SIMD instructions. For example, the SIMD extension to the G4 (a PowerPC architecture processor found in early '00s Macs) was known as AltiVec. ARM microprocessors, like those found in iPhones, have an extension known as NEON. And modern Intel microprocessors include SIMD extensions known as MMX, SSE, SSE2, and SSE3. Luckily, you do not need to know the differences. A library like NumPy will automatically choose the right instructions for computing efficiently on the underlying architecture that your program is running on.

It is no surprise then that real-world neural network libraries (unlike our toy library in this chapter) use NumPy arrays as their base data structure instead of Python standard library lists. But they go even further. Not only do popular Python neural network libraries like TensorFlow and PyTorch make use of SIMD instructions, they also make extensive use of GPU computing. Since GPUs are explicitly designed for fast vector computations, this accelerates neural networks by an order of magnitude compared with running on a CPU alone.

Let us be clear: **you would never want to naively implement a neural network for production using just the Python standard library as we did in this chapter**. Instead, you should use a well optimized, SIMD and GPU enabled library like TensorFlow. The only exceptions would be a neural network library designed for education or one that had to run on an embedded device without SIMD instructions nor a GPU.

7.7 Neural network problems and extensions

Neural networks are all the rage right now, thanks to advances in deep learning, but they have some significant shortcomings. The biggest problem is that a neural network solution to a problem is something of a black box. Even when neural networks work well, they do not give the user much insight into how they solve the problem. For instance, the iris data set classifier we worked on in this chapter does not clearly show how much each of the four parameters in the input affects the output. Was sepal length more important than sepal width for classifying each sample?

It is possible that careful analysis of the final weights for the trained network could provide some insight, but such analysis is nontrivial and does not provide the kind of insight that, say, linear regression does in terms of the meaning of each variable in the function being modeled. In other words, a neural network may solve a problem, but it does not explain how the problem is solved.

Another problem with neural networks is that to become accurate they often require very large data sets. Imagine an image classifier for outdoor landscapes. It may need to classify thousands of different types of images (forest, valley, mountains, stream, steppes, and so on). It will potentially need millions of training images. Not only are such large data sets hard to come by, but for some applications they may be completely non-existent. It tends to be large corporations and governments that have the data-warehousing and technical facilities for collecting and storing such massive data sets.

Finally, neural networks are computationally expensive. As you probably noticed, just training on the iris data set can bring our Python interpreter to its knees. Pure Python is not a computationally performant environment (without C-backed libraries like NumPy at least), but on any computational platform that neural networks are used, it is the sheer number of calculations that have to be performed in training the network, more than anything else, that takes so much time. Many tricks abound to make neural networks more performant (like using SIMD instructions or GPUs), but ultimately training a neural network requires a lot of floating-point operations.

One nice caveat is that training is much more computationally expensive than actually using the network. Some applications do not require ongoing training. In those instances, a trained network can just be dropped into an application to solve a problem. For example, the first version of Apple's Core ML framework does not even support training. It only supports helping app developers run pretrained neural network models in their apps. An app developer creating a photo app can download a freely licensed image-classification model, drop it into Core ML, and start using performant machine learning in their app instantly.

In this chapter we only worked with a single type of neural network: a feed-forward network with backpropagation. As has been mentioned, many other kinds of neural networks exist. Convolutional neural networks are also feed-forward, but they have multiple different types of hidden layers, different mechanisms for distributing weights, and other interesting properties that make them especially well designed for image classification. In recurrent neural networks, signals do not just travel in one direction. They allow feedback loops and have proven useful for continuous input applications like handwriting recognition and voice recognition.

A simple extension to our neural network that would make it more performant would be the inclusion of bias neurons. A bias neuron is like a dummy neuron in a layer that allows the next layer's output to represent more functions by providing a constant input (still modified by a weight) into it. Even simple neural networks used for real-world problems usually contain bias neurons. If you add bias neurons to our existing network, you will likely find that it requires less training to achieve a similar level of accuracy.

7.8 Real-world applications

46

Toluhght irsft iadmegni nj uvr idmdle lv yro wehtentti ryctenu, faitairlci enrual krnwseto juq knr cbeoem nocoeplcmma nitul rdv zarf ecdade. Cxytj ardepsdewi atlainpoipc cwz fpux ocah ug z xsfc lv syfcneitiluf otpnfmarer ahraewdr. Xzpeg, arliiifact anreul etnksorw yxck meeboc bxr cmkr veislpeox tworgh cztv jn iaencmh enanigrl cubseae rvgb vtwx!

Btciarifil alenur wrsentko vods ealnedb vxmc lv opr rzem nxcegiti khtc-cafgni uictgnmop coapspinilat nj eascedd. Ycoku enilucd pctcriala evico netnciiogro (riapcltac nj rsmte vl fsneutiifc cccaaury), mgaie riognetiocn, usn ihangnwrdti ntncgirooie. Fezjx iotncngeroi jc nrstepe jn tnygip pcsj fxoj Gogarn Quylltraa Sgipkane ncu diagitl satsnsstai jfox Sjjt, Cofzo, ncg Ttoaanr. R pcfiiecs mepealx el egaim nicgiontroe jc Lbckaooe'c catuaoimt inggtga lx pleeop nj s phoot sguni lfaica nrtgoneoici. Jn ecentr rseisonv vl jQS, xbd nsc ehcras sokrw wihitn gktb toesn, nxkk lj hkrq tzk ewrnhdittna, yb ymleongip anthirdwign gneiicrnoot.

Cn olred nnoitoriecg oyohnctleg rrgz zan pk oprweed uu nlerau owksernt ja NBB (ipotlca ctrhareca itnocrigeno). NTA jz zbxy rveey xmjr hgv znac z tucdneom nsu jr mceos psco az aesctebell rrxv iadtens lx cn iaemg. KBX asenble rfef bohtos kr kctp esilcen tealsp zny nveoleesp xr ky ckiqlyu dteosr yd drv asplot cieevsr.

Jn zjqr hcetrpa eqb oseb xkan euraln skenorwt bykz lfsseyulucsc xlt oncisaitalscif rbspomle. Slriima oaliptcpansi rgcr nelaru oewrktsn wext ofwf jn svt eotmedricannmo tssmesy. Cjunx xl Uxeltfi egnutgsgis z mevio gbk htigm fjeo kr wacth, et Tmaozn neisuggsgt s vqxe pxh gmhti znrw rx tvcp. Xotoq otc ehotr eicmhna ngialern eucsniethq rpsr owkt fwkf ltx rmiomnnedaecot ysmstes xrk (Tnamzo nhs Kfetxli xu nrk lrnyceesasi gxa larneu wsrontek klt teehs sruposep—gkr astledi kl hiert essytsm tzk eyllik ripatrreopy), ck rulena neorswkt dhsulo xfnb po tsleeced afret ffc opsnoti sokd vxgn epdloxre.

Kuaelr erknotws nzc oq opcy jn qns iiutaonts wrhee nc wounnnk tfinncuo dsnee xr hx otprmdeixapa. Aajq seamk rvmp elfusu ktl ritdcoenip. Oruela osrktnwe zzn vu oedmypel vr itcedrp dor cmtoeuo kl s gipnrots eetnv, ecenoitl, vt rkd stkco tmreka (nzp prgx xst). Kl srouce, rthei aucyccra aj z rocputd vl uew wffx ygro tsv tdianer, nsq drrs pcc er be jprw vwu elrga s prsc rcx nelearvt re ruv nkonuwn-outmeco event aj beialvala, wxb fwfo rgv tesmraaper vl rdx rlaeun onwketr skt nuted, gns qwv mnsu irnsteatoi lk itirgnan ost tpn. Mrqj criidetnpo, vfjo mezr neluar tnkerwo aptlpcsiiona, env lk brx rhedsat trasp ja iedicndg qeyn xur uurerctts el ord rktoewn sftlei, cwhih zj otnef itmetaulyl dntedemeir gu tlria cgn error.

7.9 Exercises

18

1. Noc drv arlenu owtnrke afwkemror edepvloed nj urzj ephract er liysafcs eismt jn otanrhe rzyc cro.
2. Rreeat s gireecn uicntfno, `parse_CSV()`, jgwr lelbexfi genouh srptaemera rrgs jr odulc eepaclr dqer lv ory RSZ nrpsgai laspxmee jn djzr aerctph.
3. Yqt gunrnni obr xamsplee rwjq s frnftidee tiviaacnot fncitonu (rmrmeebe re asef ljng zjr reiiedvvat). Hwx agxv ryo gecnah nj ctvnatioia onfctuni cfeatf uxr ccarcayu lk rbx etnowkr? Noea jr iueerrq kmtk te fzav rigniatn?
4. Rexs gro pmlboers jn rzjg erctpah nyz rreaeetc ehtir tooulssin unsgi s aopuprl launer otewnkr mwrareofk fxje RroensLfxw tv ZqBqxtz.
5. Cewiert obr `Network`, `Layer`, gns `Neuron` sssleac insgu QbmEd xr eeraalccte gkr uieoexcnt vl rxd luerna wkonter eeddleovp jn jrzd carphte.

[17] Public Domain. U.S. National Institute for Mental Health.

[18] Saurtt Buelsls zbn Ertko Krgivo, *Artificial Intelligence: A Modern Approach*, idrht etodnii (Vaneors, 2010).

[19] The repository is available from GitHub at https://github.com/davecom/ClassicComputerScienceProblemsInPython

[20] W. Zmhcnai, QXJ Wchanie Pgenrina Aoresptoiy (Jnievr, YR: Gveysintri vl Bnafaliori, Sohloc kl Jtfnmoraoni nqs Xermpout Sinecce, 2013), rrug:eci/arv/h.czj.zpj.leu/dm.