یادگیری عمیق برای انسان‌ها

درک کنید که شبکه‌های عصبی عمیق چگونه کار می‌کنند

مولف:

محیط دشپنده

ترجمه:

دکتر حمید رضا حسن نژاد مرزونی
مهندس علی جهانیان بهنمیری

فن‌آوری نوین

سرشناسه	:	دشپند، موهیت Deshpande, Mohit
عنوان و نام پدیدآور	:	یادگیری عمیق برای انسان‌ها: درک کنید که شبکه‌های عصبی عمیق چگونه کار می‌کنند/ مولف محیط دشپنده ؛ ترجمه حمیدرضا حسن‌نژادمرزونی، علی جهانیان‌بهنمیری.
مشخصات نشر	:	بابل: فناوری نوین ، ۱۳۹۹.
مشخصات ظاهری	:	۱۲۱ ص.
شابک	:	۳۷۰۰۰۰ ریال: 978-622-7393-18-7
وضعیت فهرست نویسی	:	فیپا
یادداشت	:	عنوان اصلی: Deep Learning for Human Beings : Understand How Deep Neural Networks Work.
موضوع	:	فراگیری ماشینی
موضوع	:	Machine learning
موضوع	:	شبکه‌های عصبی (کامپیوتر) -- الگوهای ریاضی
موضوع	:	Neural networks (Computer science) -- Mathematical models
شناسه افزوده	:	حسن‌نژادمرزونی، حمیدرضا، ۱۳٦٦-، مترجم
شناسه افزوده	:	جهانیان‌بهنمیری، علی، ۱۳٦۷-، مترجم
رده بندی کنگره	:	۳۲۵/۵QC
رده بندی دیویی	:	۰۰٦/۳۱
شماره کتابشناسی ملی	:	۷۳۹۵٦۰۲
وضعیت رکورد	:	فیپا

www.fanavarienovin.net

فناوری نوین بابل، کدپستی ٤٧١٦٧-٧٣٤٤٨ تلفن: ۳۲۲۵٦٦۸۷-۰۱۱

یادگیری عمیق برای انسان‌ها

ترجمه: حمید رضا حسن نژاد مرزونی، علی جهانیان بهنمیری،

نوبت چاپ: چاپ اول

سال چاپ: پاییز ۹۹

شمارگان: ۲۰۰

قیمت: ۳۷۰۰۰ تومان

نام چاپخانه و صحافی: دفتر فنی سورنا

شابک: 978-622-7393-18-7

نشانی ناشر: بابل، چهارراه نواب، کاظم‌بیگی، جنب مسجد منصور کاظم‌بیگی، طبقه اول

طراح جلد: کانون آگهی و تبلیغات آبان (احمد فرجی)

تهران، خ اردیبهشت، نبش وحید نظری، پلاک ۱۴۲ تلفکس: ۶۶۴۰۰۲۲۰-۶۶۴۰۰۱۴۴

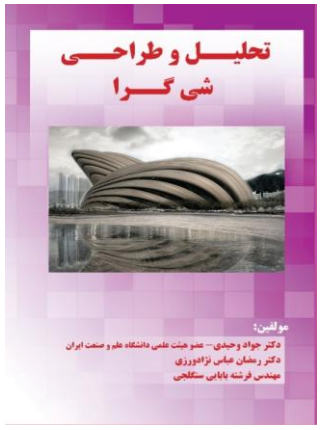
فهرست مطالب

مقدمه

الگوریتم‌های یادگیری عمیق زیرمجموعه‌ای از الگوریتم‌های یادگیری ماشین هستند که هدف آن‌ها کشف چندین سطح از بازنمودهای (نمایش)های توزیع شده از داده ورودی است. اخیرا الگوریتم‌های یادگیری عمیق زیادی برای حل مسائل هوش مصنوعی سنتی ارائه شده‌اند. در این کتاب خلاصه‌ای از چندین روش یادگیری عمیق متفاوت و پیشرفت‌های اخیر آن‌ها ارائه را می‌کنیم. شبکه عصبی عمیق در زمینه بینایی همانند دسته‌بندی تصاویر، شناسایی اشیاء، استخراج تصاویر و قطعه‌بندی معنایی دارد.

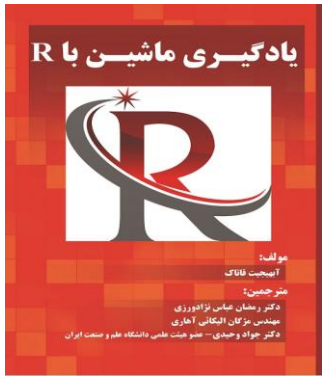
ویژگی بسیار مهم این کتاب پیاده‌سازی الگوریتم‌های مربوط به شبکه عصبی عمیق با زبان برنامه‌نویسی پایتون است.

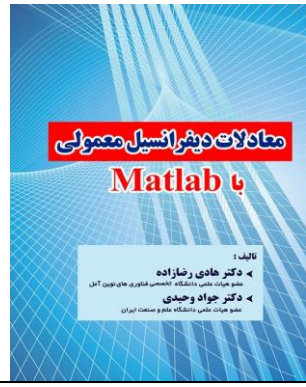
این کتاب شامل ۶ فصل است. فصل اول، به مفهوم پرسپترون‌ها (اولین شبکه‌های عصبی) پرداخته و الگوریتم-های آن را با پایتون پیاده‌سازی می‌کند. فصل دوم، راهنمای کاملی برای شبکه‌های عصبی عمیق می‌باشد و الگوریتم مربوط به شبکه‌های عصبی عمیق را با پایتون پیاده‌سازی می‌نماید. فصل سوم، شبکه‌های کانولوشن را برای بینایی ماشین شرح داده و الگوریتم‌های آن را با پایتون پیاده‌سازی می‌کند. فصل چهارم، راهنمایی برای بهبود کارایی یادگیری عمیق است که الگوریتم‌های بهبودیافته در آن با پایتون پیاده‌سازی شده است. فصل پنجم، به خودرمزگذار پرداخته، انواع خودرمزگذار را شرح می‌دهد و آن‌ها را با پایتون پیاده‌سازی می‌نماید. فصل ششم، شبکه‌های عصبی بازگشتی و شبکه‌های عصبی بازگشتی مدرن را بیان کرده، الگوریتم‌های مربوط به آن‌ها را با پایتون پیاده‌سازی می‌کند.

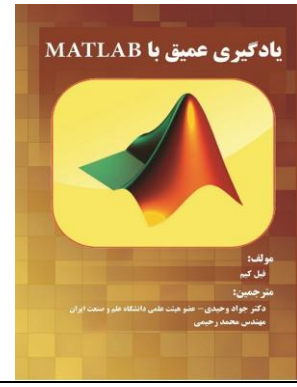

تحلیل و طراحی
شی گرا
مولفین:
دکتر جواد وحیدی- عضو هیئت علمی دانشگاه علم و صنعت ایران
دکتر رمضان عباس نژادورزی
مهندس فرشته بابایی ستکلجی


کلان داده
اصول و بهترین اقدامات
سیستم های داده‌ای بلادرنگ مقیاس پذیر
BIG
DATA
مولفین:
مارز نانان
جیمز وارن
مترجمین:
دکتر جواد وحیدی- عضو هیئت علمی دانشگاه علم و صنعت ایران
دکتر رمضان عباس نژادورزی
مهندس زهرا علیجان نژاد بابی


تجزیه و تحلیل کلان داده‌ها
فناوری‌های مخرب برای تغییر بازی
BIG
DA
مولف:
دکتر آروین ساتچی
مترجمین:
دکتر جواد وحیدی- عضو هیئت علمی دانشگاه علم و صنعت ایران
دکتر رمضان عباس نژادورزی
مهندس زهرا علیجان نژاد بابی

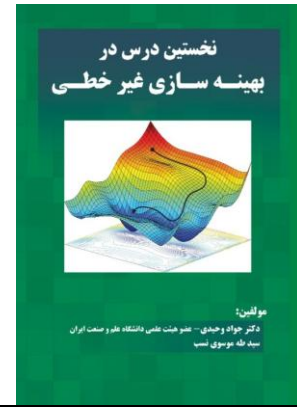

یادگیری ماشین با R
مولف:
آبیجیت فاتاک
مترجمین:
دکتر رمضان عباس نژادورزی
مهندس مژگان الیکائی آهاری
دکتر جواد وحیدی- عضو هیئت علمی دانشگاه علم و صنعت ایران


معادلات دیفرانسیل معمولی
با Matlab
تالیف :
دکتر هادی رضازاده
عضو هیات علمی دانشگاه تخصصی فناوری های نوین آمل
دکتر جواد وحیدی
عضو هیات علمی دانشگاه علم و صنعت ایران


یادگیری عمیق با MATLAB
مولف:
فیل کیم
مترجمین:
دکتر جواد وحیدی- عضو هیئت علمی دانشگاه علم و صنعت ایران
مهندس مخلد رحیمی


سوالات چهارگزینه‌ای پایتون
(۱۲۰۰ تست و پاسخ تشریحی آنها)
مولفین: مهندس یوسف عباس نژادورزی
علی رضا علی زاده حاجی
برخی از عناوین مهم
آشنایی با پایتون
ساختارهای تصمیم و حلقه‌های تکرار
توابع و ماژول‌ها
لیست، دیکشنری‌ها و رشته‌ها
کلاس‌ها و وراثت
فایل‌ها
پیمانه Tkinter
مباحث پیشرفته


یادگیری Adobe Connect 9
ایجاد و میزبانی موفقیت آمیز جلسات وب، کلاس‌های مجازی و وبینارها
با نرم‌افزار Adobe Connect
مترجمین:
دکتر جواد وحیدی - دانشگاه علم و صنعت ایران
مهندس جواد نیرگریان
برخی از عناوین مهم


نخستین درس در
بهینه سازی غیر خطی
مولفین:
دکتر جواد وحیدی- عضو هیئت علمی دانشگاه علم و صنعت ایران
سید طه موسوی نسب

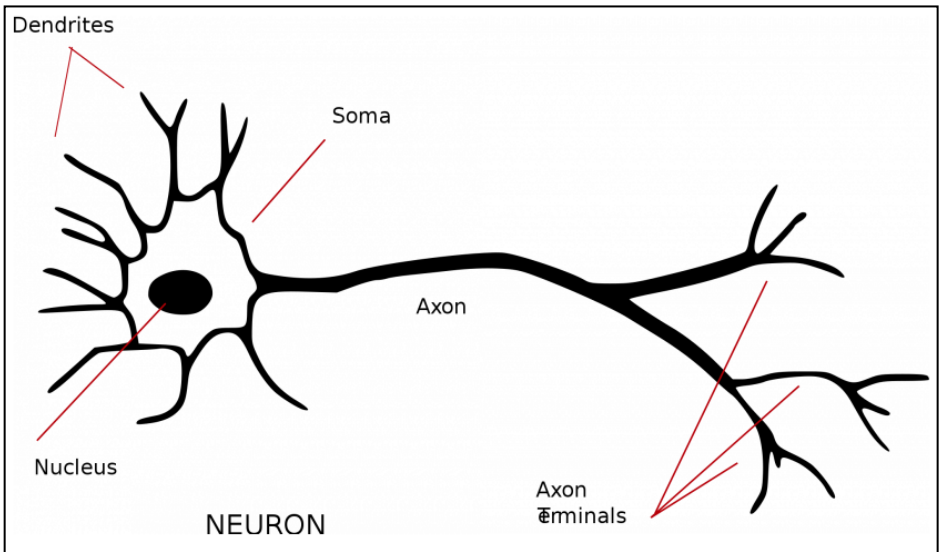
فصل ۱
پرسپترون‌ها: اولین شبکه‌های عصبی

شبکه‌های عصبی طی چند سال گذشته بسیار محبوب شده‌اند و معماری‌های جدید، انواع نورون‌ها، توابع فعال‌سازی و تکنیک‌های آموزش در تمامی اوقات در مقالات ظاهر می‌شوند. اما بدون درک اساسی از شبکه‌های عصبی، همگام‌شدن با کارهای جدید در این زمینه کاملاً دشوار است.

برای درک رویکردهای مدرن، باید کوچک‌ترین و بنیادی‌ترین بلوک ساختاری شبکه‌های عصبی عمیق اعصاب را درک کنیم: **نورون**. به‌طور خاص، خواهیم دید که چگونه می‌توان چندین نورون را در یک **لایه** ترکیب کرد و یک شبکه عصبی به نام **پرسپترون** (`perceptron`) ایجاد کرد. در این کتاب، برای ساخت یک شبکه پرسپترون، از ابتدا الگوریتم یادگیری را با کد پایتون (با استفاده از `numpy`) پیاده‌سازی می‌کنیم.

۱–۱. نورون‌های بیولوژیکی

پرسپترون‌ها و نورون‌های مصنوعی درواقع به سال ۱۹۵۸ برمی‌گردد. فرانک روزنبلات یک روانشناس بود که سعی در ارائه و تحکیم یک مدل ریاضی برای نورون‌های بیولوژیکی داشت. برای درک بهتر پرسپترون، نیاز به درک سطحی از ساختار نورون‌های بیولوژیکی در مغز خود داریم.



اجازه دهید یک نورون بیولوژیکی را در نظر بگیریم. منظور از این سلول این است که تعدادی ورودی (به شکل سیگنال‌های الکتریکی در مغز ما) را داریم، برخی پردازش‌ها را انجام داده و تعدادی خروجی (همچنین یک سیگنال الکتریکی) را تولید می‌کنیم. نکته بسیار مهمی که باید به آن توجه داشته باشید این است که ورودی‌ها و خروجی‌ها باینری هستند (۰ و ۱). یک نورون تکی از طریق **دندریت‌های** (dendrites = شاخه‌های متعدد سلول‌های عصبی) خود، معمولاً ورودی‌ها را از سایر نورون‌ها می‌پذیرد.

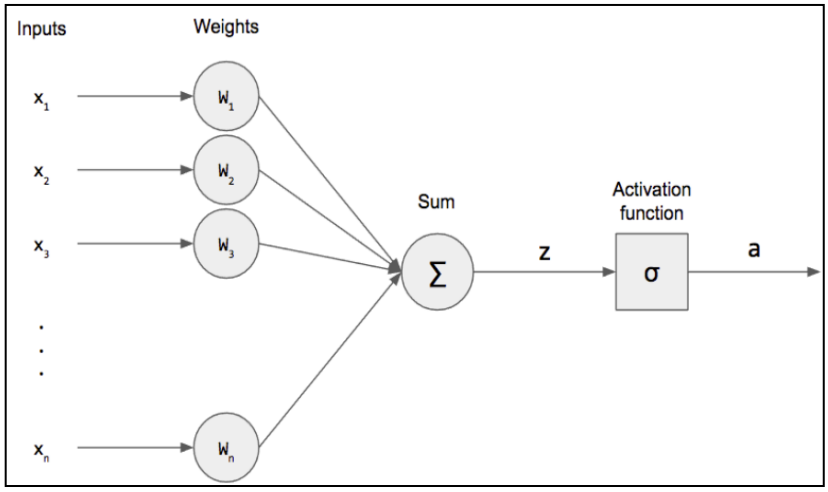
اگرچه شکل فوق آن را به تصویر نمی‌کشد، دندریت‌ها از طریق شکافی به نام **سیناپس** (synapse = محل تماس دو عصب) وزن را به یک ورودی خاص اختصاص داده با سلول‌های عصبی دیگر متصل می‌شوند. سپس، تمام این ورودی‌ها هنگامی‌که در بدن سلول یا **سوما** (**soma**) پردازش می‌شوند، با هم در نظر گرفته می‌شوند.

نورون‌ها رفتار همه‌یاهیچ را از خود نشان می‌دهند. به‌عبارت‌دیگر، اگر ترکیبی از ورودی‌ها از آستانه مشخصی فراتر رود، سپس یک سیگنال خروجی تولید می‌شود، یعنی نورون " فعال می‌شود ". اگر ترکیب از آستانه پایین بیاید، نورون هیچ خروجی ایجاد نمی‌کند، یعنی نورون " سرکوب یا مهار می‌گردد ." در مواردی که نورون برانگیخته می‌شود، خروجی در امتداد **آکسون** (axon) به **پایانه‌های آکسون** حرکت می‌کند. این پایانه‌های آکسون از طریق سیناپس به دندریت‌های نورون‌های دیگر متصل می‌شوند.

اجازه دهید لحظه‌ای به یادآوری نورون‌های بیولوژیکی بپردازیم. آن‌ها تعدادی ورودی دودویی را از طریق دندریت‌ها می‌گیرند، اما همه ورودی‌ها دارای وزن یکسان نیستند. این سیگنال‌های وزن را با هم ترکیب می‌کنیم و اگر از یک آستانه بالاتر برود، نورون برانگیخته می‌گردد. این خروجی واحد در امتداد آکسون به سلول‌های عصبی دیگر می‌رود. اکنون‌که این خلاصه را در ذهن خود داریم، می‌توانیم معادلات ریاضی را توسعه دهیم تا تقریباً یک نورون بیولوژیکی را نشان دهد.

۲–۱. نورون‌های مصنوعی

اکنون که نورون‌های بیولوژیکی را درک کرده‌ایم، مدل ریاضی باید از عملیات یک نورون پیروی کند.



در این مدل، n **ورودی** (inputs) باینری داریم (معمولاً به‌عنوان یک بردار داده می‌شود) و دقیقاً به همان تعداد **وزن** (weights) $W_1, \ldots\ldots, W_n$ نیز داریم. ورودی‌ها و وزن‌ها را در هم ضرب می‌کنیم، سپس، حاصل-ضرب‌ها را با هم جمع می‌کنیم. این را z می‌نامیم و آن را از **قبل فعال‌سازی (pre-activation)** نام‌گذاری می‌کنیم.

$$z = \sum_{i-1}^{n} W_i\, x_i = W^T x$$

(می‌توانیم این فرمول را به عنوان ضرب درونی برای پرسپترون مجدداً بنویسیم). اصطلاح دیگری وجود دارد، به نام **بایس** (bias)، که فقط یک عامل ثابت است.)

$$z = \sum_{i=1}^{n} W_i\, x_i + b = W^T x + b$$

برای راحتی در محاسبات درواقع می‌توانیم آن را در بردار وزن خود به‌صورت W. بگنجانیم و برای تمامی ورودی‌های خود ۱+ = X. را تنظیم کنیم. (این مفهوم قرار دادن بایاس در بردار وزن هنگام نوشتن کد روشن‌تر خواهد شد.)

$$z = \sum_{i=0}^{n} W_i x_i = W^T x$$

پس از محاسبه مجموع وزنی، یک تابع فعال‌سازی، σ را برای این کار اعمال و یک فعال‌سازی تولید می‌کنیم. تابع فعال‌سازی برای پرسپترون گاهی اوقات یک **تابع پله‌ای** (**step function**) نامیده می‌شود. زیرا، اگر بخواهیم آن را ترسیم کنیم، شبیه پله خواهد بود:

$$\sigma(q) = \begin{cases} 1 & q \geq 0 \\ 0 & q < 0 \end{cases}$$

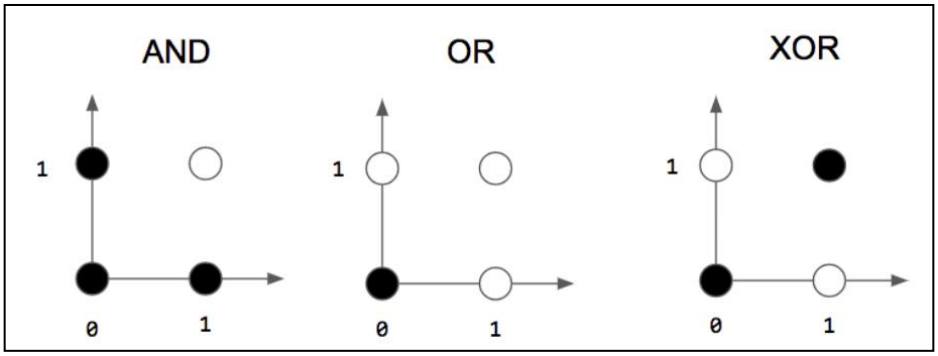
به‌عبارت‌دیگر، اگر ورودی بزرگ‌تر یا مساوی ۰ باشد، خروجی ۱ تولید می‌شود. در غیر این صورت، ۰ تولید می‌گردد. این مدل ریاضی برای یک نورون تکی (اساسی‌ترین واحد برای شبکه‌های عصبی) است.

$$a = (W^T x)$$

اجازه دهید این مدل را با نورون بیولوژیکی مقایسه کنیم. ورودی‌ها مشابه دندریت‌ها هستند و وزن‌ها سیناپس‌ها را مدل می‌سازند. ورودی‌های وزنی را با جمع‌کردن ترکیب می‌کنیم و آن جمع وزنی را به تابع فعال‌سازی می‌فرستیم. این در حالی است که ۰ به این معنی است که نورون خروجی تولید نمی‌کند، این تابع به عنوان تابع پاسخ همه‌یاهیچ عمل می‌کند. همچنین، توجه داشته باشید که ورودی‌ها و خروجی‌ها نیز باینری هستند که مطابق با مدل بیولوژیکی است.

۳–۱. توانایی‌ها و محدودیت‌های پرسپترون‌ها

ازآنجاکه خروجی یک پرسپترون باینری است، می‌توانیم از آن برای طبقه‌بندی باینری استفاده کنیم، یعنی، یک ورودی فقط به یکی از دو کلاس تعلق دارد. مثال‌های کلاسیک مورداستفاده برای توضیح آنچه که پرسپترون‌ها می‌توانند مدل کنند گیت‌های منطقی هستند!



اجازه دهید گیت‌های منطقی شکل بالا را در نظر بگیریم. یک دایره سفید به معنای خروجی ۱ و یک دایره سیاه به معنی خروجی ۰ است و محورها، ورودی‌ها را نشان می‌دهند. به‌عنوان‌مثال، وقتی ۱ و ۱ را به یک گیت AND وارد می‌کنیم، خروجی ۱، دایره سفید است. می‌توانیم پرسپترون‌هایی ایجاد کنیم که مانند این گیت‌ها عمل کنند: آن‌ها ۲ ورودی باینری را می‌گیرند و یک خروجی باینری ایجاد می‌کنند!

بااین‌حال، پرسپترون محدود به حل مسایلی هستند که به‌طور **خطی قابل‌تفکیک یا متعامد** (**linearly separable**) هستند. اگر دو کلاس ازنظر خطی قابل‌تفکیک باشند، این بدان معنی است که می‌توانیم خط واحدی ترسیم کنیم تا دو کلاس از هم جدا شوند. می‌توانیم این کار را به‌راحتی برای گیت‌های AND و OR انجام دهیم، اما هیچ خط واحدی وجود ندارد که بتواند کلاس‌ها را برای گیت XOR جدا کند! این بدان معنی است که نمی‌توانیم از پرسپترون تک لایه خود برای مدل‌سازی یک گیت XOR استفاده کنیم.

یک روش شهودی برای درک اینکه چرا پرسپترون‌ها فقط می‌توانند مسائل قابل‌تفکیک خطی را مدل کنند، نگاه کردن معادله جمع وزنی (همراه با بایاس) است.

$$\sum_{i=1}^{N} W_i\, x_i + b$$

این فرمول بسیار شبیه معادله یک خط است! (یا به‌طورکلی، یک سطح صاف بزرگ است). ازاین‌رو، در حال ایجاد یک خط هستیم و می‌گوییم که همه‌چیز در یک طرف خط متعلق به یک کلاس است و همه‌چیز

در طرف دیگر همان خط متعلق به کلاس دیگر است. این خط **مرز تصمیم‌گیری (decision boundary)** نامیده می‌شود و وقتی از یک پرسپترون تک لایه استفاده می‌شود، فقط می‌توان یک مرز تصمیم‌گیری را تولید کرد.

با توجه به این اطلاعات جدید، به نظر نمی‌رسد که پرسپترون‌ها مفید باشند! اما، در عمل، بسیاری از مسائل درواقع ازنظر خطی قابل‌تفکیک هستند. بااین‌وجود، امید برای حل مسائل غیرخطی قابل‌تفکیک از بین نرفته است! می‌توان نشان داد که سامان‌دهی چندین پرسپترون به لایه‌ها و استفاده از یک لایه میانی یا **لایه پنهان (hidden layer)** می‌تواند مشکل XOR را حل کند! این مبحث پایه و اساس شبکه‌های عصبی مدرن است!

۴-۱. کدنویسی برای پرسپترون تک لایه

اکنون‌که درک خوبی از نحوه عملکرد پرسپترون داریم، اجازه دهید یک قدم دیگر برداریم و فرمول ریاضی را به کد برنامه تبدیل کنیم. برای این منظور، از اصول شیءگرا استفاده خواهیم کرد و یک کلاس ایجاد می‌کنیم. برای ساخت پرسپترون، باید بدانیم که تعدادی ورودی برای ایجاد بردار وزن وجود دارد. دلیل این‌که یک واحد به اندازه ورودی (input_size) اضافه می‌کنیم، وجود بایاس در بردار وزن است.

```
1 import numpy as np
2
3 class Perceptron(object):
4   """Implements a perceptron network"""
5   def __init__(self, input_size):
6     self.W = np.zeros(input_size + 1)
```

همچنین باید تابع فعال‌سازی خود را پیاده‌سازی کنیم. اگر ورودی بزرگ‌تر یا مساوی 0 باشد، این تابع 1 را برمی‌گرداند، در غیر این صورت، 0 را برگشت خواهد داد.

```
1 def activation_fn(self, x):
2   return 1 if x >= 0 else 0
```

سرانجام، برای اجرا یک ورودی از طریق پرسپترون و برگرداندن یک خروجی، به یک تابع نیاز داریم. معمولاً به این تابع **پیش‌گویی (prediction)** گفته می‌شود. bias را به بردار ورودی اضافه می‌کنیم. سپس می‌توانیم ضرب درونی را به‌سادگی محاسبه کرده و تابع فعال‌سازی را اعمال کنیم.

```
1  def predict(self, x):
2    x = np.insert(x, 0, 1)
3    z = self.W.T.dot(x)
4    a = self.activation_fn(z)
5    return a
```

همه این‌ها توابع در کلاس پرسپترون هستند که برای یادگیری پرسپترون از آن‌ها استفاده خواهیم کرد.

۵–۱. الگوریتم یادگیری پرسپترون

یک پرسپترون تعریف کرده‌ایم، اما پرسپترون‌ها چگونه یاد می‌گیرند؟ روزنبلات (Rosenblatt)، خالق پرسپترون، با توجه به شهود خود در مورد نورون‌های بیولوژیکی، اندیشه‌هایی در باره نحوه آموزش نورون‌ها داشت. روزنبلات یک الگوریتم یادگیری ساده را ارائه کرد. ایده او این بود که هر نمونه ورودی را از طریق پرسپترون اجرا کنیم و اگر پرسپترون در مواقعی که لازم نیست شلیکی صورت گیرد، آن را مهار می‌کند. اگر پرسپترون در زمان لازم شلیک نکرد، آن را به هیجان بیاورید.

چگونه مهار یا تحریک کنیم؟ بردار وزن (و بایاس) را تغییر می‌دهیم!

بردار وزن پارامتری برای پرسپترون است: باید آن را تغییر دهیم تا بتوانیم به‌درستی طبقه‌بندی هر یک از ورودی‌های خود را انجام دهیم. با توجه به این موضوع، باید یک قانون به‌روزرسانی را برای بردار وزن خود بنویسیم تا بتوانیم آن را به‌طور مناسب تغییر دهیم:

$$w \leftarrow w + \Delta w$$

باید یک Δw خوبی را تعیین کنیم که آنچه را می‌خواهیم انجام دهد. ابتدا می‌توان خطا را به‌عنوان تفاوت بین خروجی مطلوب و واقعی d و خروجی پیش‌بینی‌شده y تعریف کرد:

$$e = d - y$$

توجه کنید که وقتی d و y یکسان هستند (هر دو ۰ یا هر دو ۱ هستند)، ۰ را دریافت می‌کنیم! هنگامی‌که آن‌ها متفاوت هستند، (۰ و ۱ یا ۱ و ۰) ، می‌توانیم ۱ یا ۱- را به دست آوریم. این امر به‌طور مستقیم با مهار

پرسپترون ما مطابقت دارد! این کار را با ورودی ضرب می‌کنیم تا به پرسپترون خود بگوییم بردار وزن خود را متناسب با ورودی‌مان تغییر دهد.

$$w \leftarrow w + \eta . e . x$$

یک **ابر پارامتر** (hyperparameter) η وجود دارد که به آن **نرخ یادگیری** (learning rate) گفته می‌شود. نرخ یادگیری فقط یک عامل اندازه‌گیری است که تعیین می‌کند بردار وزن چه میزان به‌روزرسانی باید داشته باشد. این یک ابرپارامتر است. زیرا توسط پرسپترون آموخته نمی‌شود (توجه کنید که هیچ قانونی برای به‌روزرسانی η وجود ندارد). اما این پارامتر را انتخاب می‌کنیم.

(برای پرسپترون‌ها، قضیه هم‌گرایی، پرسپترون می‌گوید، با توجه به این‌که کلاس‌ها بدون توجه به نرخ یادگیری، ازنظر خطی از هم جدا می‌شوند، همگرا خواهند شد. اما برای سایر الگوریتم‌های یادگیری، این یک پارامتر مهم (بحرانی) است)

اجازه دهید نگاهی دیگری به این قانون به‌روزرسانی بیندازیم. وقتی خطا ۰ باشد، یعنی خروجی همان چیزی است که انتظار داریم، پس به‌هیچ‌وجه بردار وزن را تغییر نمی‌دهیم. هنگامی‌که خطا غیر صفر است، بر این اساس بردار وزن را به‌روز می‌کنیم.

۶-۱. کدنویسی برای الگوریتم یادگیری پرسپترون

با توجه به قانون به‌روزرسانی، می‌توانیم یک تابع ایجاد کنیم تا قانون به‌روزرسانی را تا زمانی که پرسپترون بتواند همه ورودی‌های ما را به‌درستی طبقه‌بندی کند، ادامه دهیم. تا زمانی که این اتفاق نیفتد، باید از طریق داده‌ها، آموزش خود تکرار کنیم. یک دوره زمانی است که پرسپترون همه داده‌های آموزش را یک‌بار دیده باشد. معمولاً الگوریتم یادگیری خود را برای چند دوره اجرا می‌کنیم. قبل از پیاده‌سازی کد الگوریتم یادگیری، باید تغییراتی در تابع `__init__()` خود ایجاد کنیم تا نرخ یادگیری (lr) و تعداد دوره‌ها (epochs) به‌عنوان ورودی اضافه شوند.

```
1 def __init__(self, input_size, lr=1, epochs=10):
2   self.W = np.zeros(input_size+1)
3   # add one for bias
4   self.epochs = epochs
```

```
  self.lr = lr
```

اکنون می‌توانیم یک تابع ایجاد کنیم، با توجه به ورودی‌ها و خروجی‌های موردنظر، الگوریتم یادگیری پرسپترون را پیاده‌سازی کنیم. وزن خود را برای تعدادی از دوره‌ها به‌روز می‌کنیم و از طریق کل مجموعه داده آموزش را تکرار می‌کنیم. هنگام انجام به‌روزرسانی وزن، بایاس را وارد ورودی می‌کنیم. سپس، می‌توانیم پیش‌بینی خود را ایجاد کنیم، خطای خود را محاسبه کرده و قانون به‌روزرسانی خود را انجام دهیم.

```
def fit(self, X, d):
  for _ in range(self.epochs):
    for i in range(d.shape[0]):
      y = self.predict(X[i])
      e = d[i] - y
      self.W = self.W + self.lr * e * np.insert(X[i], 0, 1)
```

کد کامل کلاس مربوط به پرسپترون در زیر آمده است:

```
class Perceptron(object):
  """Implements a perceptron network"""
  def __init__(self, input_size, lr=1, epochs=100):
    self.W = np.zeros(input_size+1)
    # add one for bias
    self.epochs = epochs
    self.lr = lr

  def activation_fn(self, x):
    #return (x >= 0).astype(np.float32)
    return 1 if x >= 0 else 0

  def predict(self, x):
    z = self.W.T.dot(x)
    a = self.activation_fn(z)
    return a

  def fit(self, X, d):
    for _ in range(self.epochs):
      for i in range(d.shape[0]):
        x = np.insert(X[i], 0, 1)
        y = self.predict(x)
        e = d[i] - y
        self.W = self.W + self.lr * e * x
```

اکنون‌که کد پرسپترون خود را کامل کرده‌ایم، می‌توانیم برخی از داده‌های آموزشی را به آن ارائه دهیم و ببینیم که این کد کار می‌کند! یک مجموعه ساده داده، گیت AND است. در زیرمجموعه‌ای از ورودی‌ها و خروجی‌ها آمده‌اند:

```
if __name__ == '__main__':
  X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
  ])
  d = np.array([0, 0, 0, 1])

  perceptron = Perceptron(input_size=2)
  perceptron.fit(X, d)
  print(perceptron.W)
```

فقط در چند خط، می‌توانیم از پرسپترون استفاده کنیم! در پایان، بردار وزن را چاپ می‌کنیم. با استفاده از داده‌های گیت AND، باید یک بردار وزن [۱ , ۲ , ۳-] را به دست آوریم. بدین معنی که بایاس (bias) ۳- است و برای X_1 و X_2 وزن به ترتیب ۱ و ۲ است.

برای تأیید صحت این بردار وزن، می‌توانیم چند مثال را بررسی کنیم. اگر هر دو ورودی ۰ باشند، قبل از فعال‌سازی ۳- =۱*۰+۲*۰+۳- خواهد بود. هنگام اعمال تابع فعال‌سازی، ۰ را دریافت می‌کنیم که دقیقاً ۰ و ۰ است! می‌توانیم این را برای سایر گیت‌ها نیز امتحان کنیم. توجه داشته باشید که این تنها بردار وزن درست نیست.

ازنظر فنی، اگر یک بردار وزن تکی وجود داشته باشد که بتواند کلاس‌ها را از هم تفکیک کند، تعداد نامحدودی از بردارهای وزن وجود دارند. کدام بردار وزنی که به دست می‌آوریم بستگی به مقداردهی اولیه بردار وزن دارد.

به‌طور خلاصه، پرسپترون‌ها ساده‌ترین نوع شبکه عصبی هستند: آن‌ها ورودی می‌گیرند ، هر ورودی را با وزن ضرب می‌کنند، مجموع ورودی‌های وزنی را می‌گیرند و یک تابع فعال‌سازی را اعمال می‌کنند. ازآنجاکه آن‌ها توسط نورون بیولوژیکی توسط فرانک روزنبلات مدل‌سازی شده‌اند، آن‌ها فقط مقادیر دودویی را می-

گیرند و تولید می‌کنند. به‌عبارت‌دیگر، می‌توانیم طبقه‌بندی باینری را با استفاده از پرسپترون انجام دهیم. یکی از محدودیت‌های پرسپترون‌ها این است که آن‌ها فقط می‌توانند مسائل قابل‌تفکیک خطی را حل کنند. در دنیای واقعی، بسیاری از مسائل به‌صورت خطی قابل‌تفکیک هستند. به‌عنوان‌مثال، می‌توانیم از یک پرسپترون برای تقلید از گیت AND یا OR استفاده کنیم. بااین‌حال، از آنجا که XOR به‌صورت خطی قابل‌تفکیک نیست، نمی‌توانیم از پرسپترون‌های تک لایه برای ایجاد یک گیت XOR استفاده کنیم. الگوریتم یادگیری پرسپترون متناسب با شهود روزنبلات است: اگر نورون در مواقعی که لازم نیست شلیک کند، مهار می‌کند و اگر نورون در زمان شلیک، شلیک نمی‌کند، آن را تحریک می‌کند. می‌توانیم از این اصل ساده استفاده کرده و یک قانون به‌روزرسانی برای وزن‌های خود ایجاد کنیم تا به پرسپترون خود توانایی یادگیری بدهیم.

پرسپترون‌ها پایه و اساس شبکه‌های عصبی هستند، بنابراین درک خوب از آن‌ها در هنگام یادگیری درباره شبکه‌های عصبی عمیق سودمند خواهد بود!

فصل ۲ راهنمای کامل برای شبکه‌های عصبی عمیق

۲–۱. راهنمای کامل برای شبکه‌های عصبی عمیق : بخش اول

شبکه‌های عصبی ده‌ها سال است که وجود دارد، اما موفقیت اخیر، ناشی از توانایی برای آموزش موفقیت‌آمیز آن‌ها با لایه‌های پنهان است. در این فصل جعبه سیاه را که شبکه‌های عصبی عمیق است، باز خواهیم کرد و به چندین الگوریتم مهم برای درک نحوه کار آن‌ها می‌پردازیم. برای درک بهتر، یک شبکه عصبی عمیق را از ابتدا کد نویسی می‌کنیم و آن را با یک مجموعه داده مشهور آموزش می‌دهیم. کدها و مجموعه داده را می‌توانید از آدرس زیر دانلود کنید:

https://pythonmachinelearning.pro/complete-guide-to-deep-neural-networks-part-/1

۲–۱–۱. ارقام دست‌نویس : مجموعه داده MNIST

برای ایجاد انگیزه در بحث درباره شبکه‌های عصبی، اجازه دهید به مسئله تشخیص رقم دست‌نویس نگاهی بیندازیم.

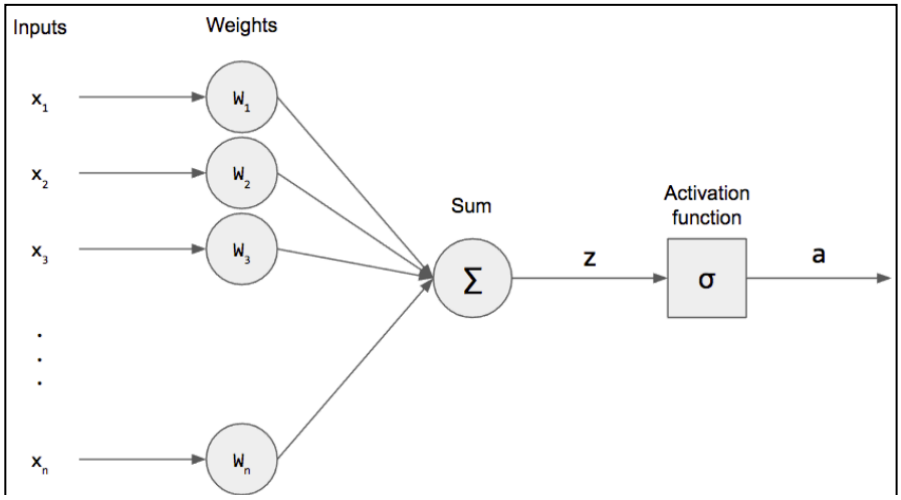
هدف این است که رقم صحیح دست‌نویس (۹–۰) داده‌شده را مشخص کنید. به‌عبارت‌دیگر، می‌خواهیم تصاویر را به ۱۰ کلاس طبقه‌بندی کنیم (یک کلاس برای هر رقم). این چالش‌برانگیزتر از آن است که فکر کنیم : هر رقم دست‌نوشته جدید می‌تواند تغییرات کمی را در خود داشته باشد، بنابراین استفاده از نمایش ثابت / استاتیک یک رقم دست‌نویس، صحت خوبی نخواهد داشت. بااین‌حال، یادگیری ماشین مبتنی بر داده(data-driven) است و می‌توانیم از آن برای حل مشکل خود استفاده کنیم. به‌طور خاص، از شبکه‌های عصبی استفاده خواهیم کرد.

خوشبختانه، نیازی به جمع‌آوری داده‌ها نیست. در حقیقت، یک مجموعه داده بسیار مشهور وجود دارد، به‌صورت محاوره‌ای به نام MNIST، که از آن استفاده خواهیم کرد. در روزهای اول، از آن برای آموزش اولین شبکه عصبی کانولوشن مدرن استفاده می‌شد. هنوز هم گاهی اوقات به‌عنوان یک مجموعه داده برای آموزش و نمایش نتایج استفاده می‌شود. در حقیقت، هنوز چالش‌هایی برای دست‌یابی به بهترین دقت وجود دارد! در زمان نوشتن این کتاب، نتایج پیشرفته در این مجموعه داده، با دقت ۹۹.۷۹ ٪ بود!

مجموعه داده را در فایل زیپ قرار داده‌ام، و تصویر فوق نمونه‌ای است که از این مجموعه داده گرفته‌شده است. برای ارائه اطلاعات بیش‌تر در مورد مجموعه داده‌ها، تصاویر باینری از یک رقم دست‌نوشته (۹-۰) به ابعاد ۲۸×۲۸ تشکیل‌شده‌اند. **مجموعه آموزش** (**training set**) فراهم‌شده (داده‌هایی که برای آموزش شبکه خود استفاده می‌کنیم) دارای ۶۰۰۰۰ تصویر است و **مجموعه تست (testing set)** (داده‌ای که برای ارزیابی شبکه استفاده می‌کنیم) ۱۰۰۰۰ تصویر دارد. از آن برای شبکه عصبی استفاده خواهیم کرد و نتایج خود را با مدرن‌ترین سطح مقایسه خواهیم کرد.

۲–۱–۲. پرسپترون‌های تک لایه Recap

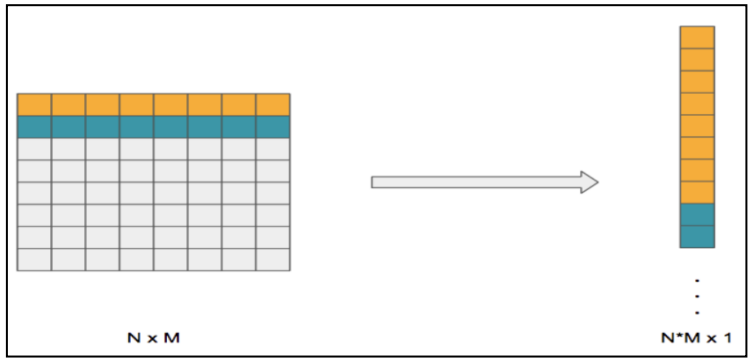
قبل از شروع بحث در مورد پرسپترون‌های باز چندلایه، اگر در حال حاضر با پرسپترون‌ها آشنا نیستید، توصیه می‌کنم فصل قبل را بخوانید تا با این مدل‌ها آشنا شوید، زیرا از این شبکه‌های عصبی کوچک شروع خواهیم کرد و پیچیدگی بیش‌تری را به آن اضافه می‌کنیم. برای جذب سریع پرسپترون‌های تک لایه، یک نورون دارای ساختار گرافیکی شبیه شکل زیر بود:



مجموع وزنی (به‌اضافه یک اصطلاح بایاس‌ها) را از ورودی خود می‌گیریم و یک تابع فعال‌سازی غیرخطی اعمال می‌کنیم. ازنظر ریاضی می‌توانیم عبارات زیر را بنویسیم:

$$z = \sum_{i=0}^{N} W_i\, x_i = W^T x$$
$$a = \sigma(z)$$

به یاد داشته باشید که z به‌عنوان **پیش‌فعال‌سازی (pre-activation)** خوانده می‌شود و a **پس‌فعال‌سازی (post-activation)** است. زیرا تابع فعال‌سازی را اعمال کردیم. برای مجموعه داده MNIST، یک تصویر داریم و نه بردار ورودی. بنابراین، چگونه می‌توانیم یک تصویر 2D را به یک بردار 1D تبدیل کنیم؟ به‌راحتی آن را تخت می‌کنیم!

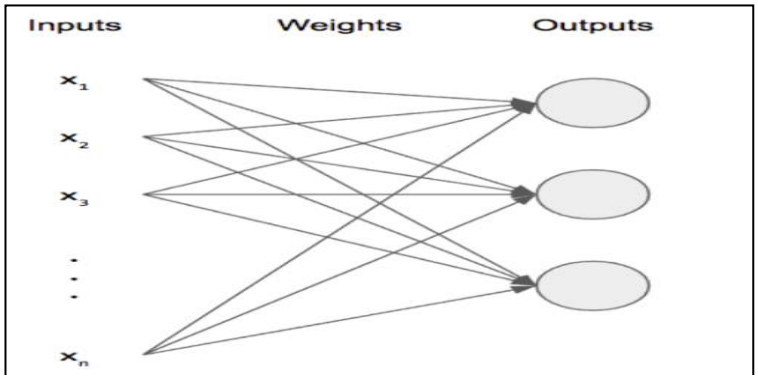


ردیف دوم تصویر خود را می‌گیریم و آن را به انتهای ردیف اول اضافه می‌کنیم. ردیف سوم را به انتهای دو ردیف اول متصل شده اضافه می‌نمایم و این روند را ادامه می‌دهیم. آنچه در پایان داریم یک بردار 1D از تصویر 2D ما است. به‌این‌ترتیب ورودی‌های خود را به شبکه عصبی وارد خواهیم کرد. به‌طور خاص، برای مجموعه داده MNIST، تصاویر از ۲۸*۲۸ در یک بردار تکی ۱ * ۷۸۴ = ۱*۲۸*۲۸ که اندازه لایه ورودی ما است، تخت می‌شوند.

(نکته منفی این رویکرد این است که اطلاعات مکانی را از دست می‌دهیم. یک نوع شبکه عصبی برای تصاویر به نام یک **شبکه عصبی کانولوشنال** (convolutional neural network) وجود دارد که در آن تصویر ورودی را تخت نمی‌کنیم. این شبکه‌های عصبی معمولاً در انجام کارهای تصویری بهتر از شبکه‌های عصبی معمولی عمل می‌کنند.

۳-۱-۲. نورون‌های با خروجی چندگانه

قبل از این‌که این مورد را به چندین لایه گسترش دهیم، ابتدا این مورد را به چندین خروجی توسعه می‌دهیم. در حال حاضر، فقط یک خروجی واحد داریم: الف. گرچه این خروجی برای طبقه‌بندی باینری مفید است، اما برای موارد دیگر مفید نیست. اگر می‌خواستیم شبکه‌ای ایجاد نماییم که بتواند بیش از دو کلاس را طبقه‌بندی کند، باید نورون‌های با خروجی بیش‌تری اضافه کنیم. برای مجموعه داده‌های MNIST، از آنجا که ۱۰ کلاس داریم (برای هر رقم یک کلاس)، به نورون با ۱۰ خروجی نیاز داریم. (برای شفافیت شکل، فقط نورون با ۳ خروجی نشان داده‌ام.)



وقتی این کار را انجام می‌دهیم، بعضی از محاسبات تغییر می‌کند: بردار وزن (**weight vector**) به یک ماتریس W (**matrix W**) با ابعاد 3*N تبدیل می‌شود. یا به‌طورکلی اندازه لایه خروجی × اندازه لایه ورودی. علاوه بر این، قبل و بعد از فعال‌سازی‌های بردارهای z و a می‌باشند. تابع فعال‌سازی σ فقط به‌صورت عنصر محور (element-wise) بر روی هر یک از قطعات بردار z اعمال می‌شود. زیبایی این تعمیم این است که می‌توانیم معادلات خود را دوباره بنویسیم، و وقتی‌که یک نورون خروجی واحد داشته‌ایم، آن‌ها تقریباً یکسان به نظر می‌رسیدند.

در حال حاضر در بعضی از جاها به‌جای بردارها، اسکالرها و به‌جای ماتریس‌ها در برخی جاهای دیگر بردارها را داریم.

$$z = Wx + b$$
$$a = \sigma(z)$$

(بایاس را صریحاً گنجانده‌ام و از این به بعد به این کار ادامه خواهم داد).

می‌توانیم این روابط را به صورت اسکالرها نیز بنویسیم. فرض کنید اندیس لایه ورودی j، اندیس لایه خروجی k است، می‌توانیم فرمول بالا را به‌صورت زیر بازنویسی کنیم.

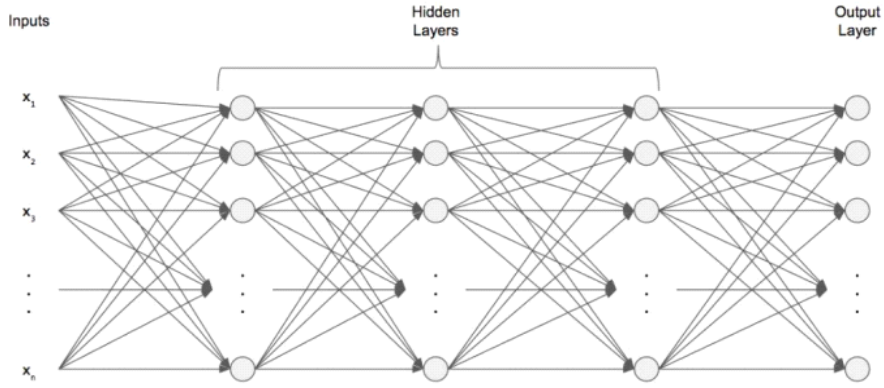
$$Z_k = \sum_j W_{kj}\, x_k + b_k$$
$$a_k = \sigma(z_k)$$

این فرمول فقط می‌گوید که برای به دست آوردن پیش فعال‌سازی یک نورون k دلخواه در لایه خروجی، باید هر ورودی را بگیریم، آن را با وزنی که آن ورودی را به آن نورون متصل می‌کند، ضرب کنیم و بایاس آن نورون را اضافه کنیم. برای این‌که متقاعد شوید که این فرم مؤلفه – محور (component-wise) صحیح است، شکل بالا را ببینید و فقط اولین نورون خروجی، یعنی k = 1 را در نظر بگیرید. این فرم مولفه بعدهاً هنگامی که درباره پس‌انتشار بحث می‌کنیم، مفید خواهد بود. با نوشتن فرم‌های مؤلفه شروع خواهیم کرد و سپس با استفاده از جبر خطی در کد، بردارسازی می‌نمایم. در حالت کلی، به خاطر کتابخانه‌های بی‌شمار (به‌عنوان‌مثال، numpy) که مخصوص کد بردارسازی شده ایجاد گردیده‌اند،کد بردارسازی شده نسبت به کد بردارسازی نشده تمایل به اجرای سریع‌تر دارد.

اما چگونه می‌توانیم داده‌های خود(ورودی‌ها و خروجی‌های **مقدار واقعی(ground-truth)**) را جهت یادگیری چند طبقه ساختار دهیم؟ ورودی تغییر نمی‌کند، اما، وضعیت واقعی را می‌گیریم و آن را به‌عنوان یک بردار وان هات (one-hot)رمزگذاری می‌کنیم. یک بردار با همان طول کلاس ایجاد می‌کنیم و یک "1" را در مکانی قرار می‌دهیم که مطابق با کلاس صحیح باشد. مجموعه داده‌های MNIST ما را در نظر بگیرید. اگر یکی از ورودی‌های ما واقع ۴ باشد پس بردار مقدار واقعی ما $[0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0]^T$ خواهد بود. بردار، یک بردار صفر است، به‌جز این که یک "1" را در مکان کلاس صحیح داریم (موقعیت شروع معادل با رقم صفر است). ازآنجاکه تعداد نورون‌های خروجی با تعداد کلاس‌ها مطابقت دارد، طول بردار همیشه برابر با تعداد خروجی نورون‌ها است.

۴-۱-۲. فرمول پرسپترون چندلایه

اکنون‌که می‌دانیم چگونه می‌توان چندین نورون خروجی را ایجاد کرد، درنهایت می‌توانیم برای **شبکه عصبی عمیق (deep neural network)** یا همان پرسپترون چندلایه **(multilayer perceptron)** به فرمولاسیون برسیم.



هنوز یک لایه ورودی و یک لایه خروجی داریم، هرکدام تعداد زیادی از سلول‌های عصبی موردنیازمان را دارند. اما در بین آن‌ها، تعدادی **لایه پنهان** (**hidden layers**) داریم که هرکدام تعدادی نورون دارند. این‌ها **لایه‌های پنهان** نامیده می‌شوند، زیرا، به‌طور مستقیم با جهان خارج ارتباط ندارند. آن‌ها از جهان خارج

پنهان‌شده‌اند. هر نورون در یک لایه پنهان به هر نورون در لایه پنهان بعدی متصل است. در شکل بالا سه لایه پنهان داریم، بنابراین این شکل یک شبکه عصبی ۴ لایه‌ای است. وقتی تعداد لایه‌ها را می‌گوییم، لایه ورودی را در نظر نمی‌گیریم. زیرا درواقع **"لایه"** نیست. این هم‌جایی است که قسمت عمیق شبکه‌های عصبی عمیق وارد می‌شود. شبکه‌های عمیق لایه‌های پنهان بسیاری دارند!

بنابراین، شبکه‌های عصبی عمیق چگونه کار می‌کنند؟ آن‌ها به‌صورت تکراری کار می‌کنند: با توجه به ورودی، مجموع وزنی (به‌علاوه بایاس) را حساب می‌کنیم و تابع فعال‌سازی را اعمال می‌کنیم. برای لایه مخفی بعدی، بعد از فعال‌سازی لایه قبل از آن، لایه پنهان ۱ را می‌گیریم و با استفاده از یک ماتریس با وزن متفاوت، مجموع وزنی(به‌علاوه بایاس) را گرفته و تابع فعال‌سازی را اعمال می‌کنیم. سپس تکرار می‌کنیم تا به لایه خروجی برسیم. به همین دلیل به شبکه‌های عصبی، **شبکه‌های پیش‌خور**(feedforward networks) نیز گفته می‌شوند: ورودی خود را از طریق شبکه به جلو هدایت می‌کنیم. خروجی‌های یک لایه پنهان به ورودی‌های لایه مخفی بعدی تبدیل می‌شوند.

از لحاظ ریاضی، باید اسکریپت دیگری را برای نشان دادن این که به کدام لایه مراجعه می‌نماییم، اضافه کنیم:

$$z^{(l)} = W^{(l)}a^{(l-1)} + b^{(l)}$$
$$a^{(1)} = \sigma(z^{(l)})$$

که در آن $a^{(1)} = x$ و $l \in [2, L]$، یعنی، لایه‌های L وجود دارد. البته می‌توانیم بنویسیم که این فرم قطعه نیز است.

درباره توان‌ها صحبت نخواهیم کرد. L فقط به معنای لایه است. همچنین توجه داشته باشید که در پیش فعال‌سازی، فقط به x اشاره نمی‌کنیم، بلکه در محاسبه پیش فعال‌سازی لایه L، از لایه قبلی L-1 استفاده می‌کنیم.

۵–۱–۲. آموزش شبکه عصبی با نزول (کاهش) گرادیان

اکنون‌که شبکه عصبی خود را شکل داده و ساخته‌ایم، اجازه دهید ببینیم چگونه می‌توانیم آن را آموزش دهیم. برای آموزش شبکه عصبی خود، به روشی نیاز داریم تا بتواند کارایی شبکه با وزن و بایاس فعلی را