:~$ mateusz puto

Notatnik
Sztucznej
Inteligencji

Spis treści

Rozdział 1

Wstęp

1.1 Cele...

Cel jest intuicyjnie zdefiniowany jako pragnienie, życzenie lub coś w kierunku czego dążymy. W codziennym życiu często mówimy o naszych celach. Są one tym, wokół czego skupiają się nasze myśli i rozmowy. Dzieje się tak, ponieważ cele ze względu na to, że nie możemy ich zrealizować od razu, są związane z pewnym komponentem czasowym. To znaczy, że nasze cele nie zmieniają się wraz z upływem czasu. Dokładniej mówiąc, cele nadrzędne nie ulegają zmianie, podczas gdy cele podrzędne, służące realizacji tych ważniejszych dla nas celów, są dostosowywane do zmieniającej się sytuacji. Mówimy np. że naszym celem jest zrobienie lub osiągnięcie czegoś, co sądzimy, że przyniesie nam pewną korzyść, lecz często nie sama korzyść jest tym, czego pragniemy, a raczej dążenie do zrealizowania celu nadrzędnego, który ma dla nas ponadprzeciętną wartość. Ze względu na długotrwałość i stosunkową rzadkość występowania, pewne cele są nam bliskie, jesteśmy z nimi zżyci i utożsamiamy się z nimi. Mimo tego istnieje też rodzaj celów, których używamy instrumentalnie. Te cele służą nam zazwyczaj tylko przez krótki czas, występują częściej i są jedynie wykonawcami celów nadrzędnych. Tak samo, jak my mamy rozbudowane relacje i metody radzenia sobie z celami, w podobny sposób powinno się to odbywać w dziedzinie sztucznej inteligencji. Jednak, ze względu na ogromne skomplikowanie naszych metod wyszukiwania i rozwiązywania problemów, próba imitacji ludzkiego zachowania wydaje się ponad nasze siły. Pomyślmy o tym, że przecież nawet dokładnie nie wiemy jakie procesy biorą udział w naszym własnym rozumowaniu. Czy w chwili kiedy mamy moment 'Aha' wiemy jakie procesy nas do niego doprowadziły? Także nauka pozostawia nas z ograniczonymi metodami badania mózgu z powodu jego odizolowania w czaszce. Pomimo wprowadzenia nowych metod obrazowania, opracowywanych coraz to nowych leków działających na mózg, czy operacji na nim, wciąż niewiele wiadomo o sposobach jego działania. Nie jesteśmy także jeszcze pewni, w jaki sposób przełożyć to, co wiemy na algorytmy, a to jest coś, co będzie nas najbardziej interesować w kontekście dziedziny sztucznej inteligencji. W konsekwencji nie pozostaje nam nic innego jak pójść drogą matematyki i spróbować nieco sformalizować znaczenia słowa „cel". Tak samo, jak my mamy, często burzliwe, relacje z naszymi celami, tak samo powinniśmy podać osobę, rzecz czy byt, który będzie

odpowiedzialny za realizowanie zadanego mu przez nas celu. Ten byt będzie w specjalnej relacji ze swoim celem, to znaczy połączymy go z nim raz na zawsze, tak aby w każdej chwili był skupiony na swoim celu. Takie połączenie przypomina małżeństwo, i tak jak małżeństwo taką relację można zerwać, lecz nie obejdzie się to bez pewnych perturbacji. Może się okazać że jeśli zamienimy cel, to nasz byt nie będzie go realizował tak samo dobrze. Może się też okazać, że będzie go realizować w sposób lepszy niż losowy, co będziemy mogli wykorzystać. Nie wnikając teraz jednak w szczegóły, skupimy się na definicji. Wykonawcę, inaczej mówiąc, istotę, która posiada pewien cel, nazwiemy zgodnie z literaturą RL, **aktorem** (ang. agent). Dalej, aby pokazać rzeczywiste działanie prezentowanych systemów, będziemy wprowadzać co jakiś czas definicje. Powiedzieliśmy, że do realizowania celu jest nam potrzebny aktor, jak go nazwaliśmy. Na razie posługiwaliśmy się mglistą definicją samego celu. Teraz podamy bardziej formalną definicję: Cel określimy jako wartość numeryczną, która jest dostępna dla aktora w pewnych momentach czasu. Aktor chce również, aby ta wartość była jak największa. Jest to jednoznaczne z matematyczną definicją maksymalizacji, więc możemy zapisać, że:

$$v = max(z) \tag{1.1}$$

Gdzie $\boldsymbol{v}$ jest celem, a $\boldsymbol{z}$ jest zmienną wartością, która zależy od pewnych znanych lub nieznanych czynników. $\boldsymbol{V}$ nazwiemy funkcją wartości.

Funkcja wartości (ang. value function) jest pewną funkcją, która określa, w jakim stopniu aktor zrealizował swój cel. Mówi nam ona, ile nagrody otrzyma on w wyniku swoich działań i jak dobrze radzi on sobie w danej sytuacji, przypisując jakąś liczbę, określającą jak bardzo dana sytuacja jest pożądaną, do każdej możliwej sytuacji. Oczywiście funkcja wartości może się zmieniać wraz z upływem czasu i ta zmiana będzie nam mówić o poprawie lub pogorszeniu sytuacji aktora. O celu możemy równocześnie myśleć intuicyjnie tak, jak w codziennym życiu. Tu celem może być bogactwo, miłość czy też wiedza, a odpowiadającą tym celom funkcją wartości może być kolejno ilość pieniędzy na koncie, siła uczucia drugiej osoby i ilość przeczytanych książek. Dostęp do funkcji wartości możemy sobie wyobrazić jako odczuwanie, tak jak sami możemy odczuwać czy jest nam dobrze, czy źle, zimno czy ciepło itd.

Podajmy jeszcze jeden przykład:

Powiedzielibyśmy, że celem istoty w ujęciu darwinistycznym jest przetrwanie i reprodukcja. Tak więc spróbujmy to zapisać formalnie za pomocą naszej definicji.

$$v = max(p + k * r) \tag{1.2}$$

Gdzie $\boldsymbol{p}$ jest zmienną związaną z przetrwaniem (wyrażoną np. w latach życia), $\boldsymbol{r}$ jest zmienną związaną z reprodukcją (np. liczba potomstwa) i ponieważ możemy cenić potomstwo bardziej lub mniej niż rok życia dodajemy kurs wymiany $\boldsymbol{k}$ między tymi dwoma zmiennymi. Równanie (1.2) pokazuje jeden z możliwych celów, którymi mogą kierować się aktorzy. W ciągu czytania tego tekstu poznamy różne inne cele, które mogą zostać użyte w innych okolicznościach.

1.2 ... i jak je osiągnąć

Powiedzieliśmy, że realizatorem celów są aktorzy. Jednak ta definicja pomija najważniejsze pytanie, a mianowicie: Jak aktorzy realizują swoje cele? Pod tym problemem kryje się najwięcej trudności i to jest właśnie główny cel sztucznej inteligencji (albo krócej: AI od ang. artificial intelligence), jakim jest tworzenie aktorów, którzy osiągają jakiś cel, a więc maksymalizują jakąś zadaną wartość. Często ci, którzy tworzą takie rozwiązania nie myślą nawet o aktorach, bo rozumieją bez zastanawiania się, że potrzebują pewnego modelu rzeczywistości, który będzie wyjaśniał działanie systemu, którym się zajmują. Takim modelem będą właśnie procesy myślowe aktora. Tak więc główny nacisk jest położony na osiąganie jasno zdefiniowanych celów, czyli takich, które da się mierzyć, za pomocą komputerów. To bardzo ogólne podejście, za pomocą którego możemy próbować rozwiązać wszelakie ciężkie problemy. Jedynymi ograniczeniami są nasza zdolność zdefiniowania problemu i umiejętność przekazania rozwiązania w języku zrozumiałym dla komputerów. Oczywiście możemy też nie znać rozwiązania problemu, który zdefiniowaliśmy i wtedy dodatkową trudnością jest to, że nasz system musi być w stanie znaleźć zadowalające nas rozwiązanie. Znajdywanie tych rozwiązań, będzie najważniejszą częścią wiedzy o systemach sztucznej inteligencji. W pewnym więc sensie badanie AI jest nauką nakierowaną na rozwiązanie wszystkich istniejących problemów. W pierwszej chwili może wydawać się to nam myśleniem życzeniowym albo nawet, że jest to nieosiągalne, przecież jeśli istniałby prosty, definiowalny matematycznie sposób radzenia sobie z każdą napotykaną sytuacją to na pewno ktoś odkryłby go już dawno temu. Jednak co ciekawe, mimo iż pewne intuicje na temat działania mózgu były obecne od dawna, to próby systematycznego opisu jego działania rozpoczynają się w XIX w. w psychologii i na przełomie wieków w matematyce jako logika. W następującym potem okresie psychologia posłużyła jako inspiracja, do badania procesów występujących w mózgu a logika dostarczyła, poza systemem rozumowania, podstawy do stworzenia głównego obiektu do przeprowadzania eksperymentów, jakim okazał się komputer. Mimo iż rozwiązania tu opisane korzystają szczodrze z rozwiązań znanych w matematyce od wieków, to mogły one zostać zastosowane dzięki rozległej mocy obliczeniowej dostarczonej przez nowoczesne komputery. Nie oznacza to, jednak że wszystko było gotowe i czekało na zastosowanie. Chociażby sieci neuronowe, według naszej wiedzy, nie śniły się nikomu przed wynalezieniem komputerów.

Jest to bardzo ciekawe gdyż model sieci neuronowej lub chociażby samego neuronu, zdaje się bardzo prosty, po jego zrozumieniu. Okazuje się, że nasza wyobraźnia mimo swojej mocy jest ograniczona, więc większość rozwiązań, o których tu mowa zostało opracowanych dopiero w ciągu kilkudziesięciu ostatnich lat. Właściwie, jak zobaczysz w ostatnim rozdziale, ten proces nadal trwa. Prawdopodobnie zostało nam jeszcze wiele do odkrycia, zanim będziemy mogli uznać problem za całkowicie rozwiązany. A jeśli chodzi o nadmierną łatwość takiego podejścia do AI, to jak to bywa, wiele rozwiązań, które mogą się wydawać łatwe na początku, okazuje się niezmiernie skomplikowane i trudne w wykonaniu. Chociaż dłuto może ociosać kamień w dowolnym kształcie, to stworzenie katedry za pomocą dłuta jest wielkim wyzwaniem. Zobaczymy, iż formalizowanie pewnych intuicyjnych koncepcji może być zawiłe i że dyscyplina sztucznej inteligencji jest w istocie dziedziną inżynieryjną.

1.3 Właściwy wstęp

Powiedzieliśmy już, czym zajmuje się dziedzina sztucznej inteligencji, jednak nie wspomnieliśmy, że bardzo podobną dziedziną wykorzystującą komputery do rozwiązywania problemów jest informatyka. Jaka jest różnica między dziedziną informatyki a dziedziną sztucznej inteligencji? Sztuczna inteligencja wywodzi się z informatyki na wiele sposobów. Zazwyczaj ci sami ludzie pracujący w wydziałach informatyki zajmowali się też problemami sztucznej inteligencji, używali do tego znanych przez siebie metod, czyli metod przetwarzania informacji. Czytelnik mógłby nawet powiedzieć, że rozwiązania, które kiedyś uważane były za AI, obecnie uchodzą za zwyczajne algorytmy. Jednak zwyczajową datą uznawaną za początek AI jest rok 1956, kiedy to odbył się tzw. warsztat w Dartmouth. W nocie proponującej ten warsztat napisano m.in. że proponujący warsztat, czyli McCarthy, Minsky, Rochester i Shannon, nazwiska ważne dla dziedziny, myślą, że dziesięć osób pracujących przez lato jest w stanie dokonać znaczących postępów. Pomimo wielkich planów efektem były raczej przyjaźnie i ekscytacja nowymi możliwościami niż realne postępy. Jak więc sami uczestnicy tego warsztatu definiują pojęcie AI? John McCarthy: [AI to] „nauka i inżynieria tworzenia inteligentnych maszyn". Moglibyśmy podejść do sprawy w ten sam sposób, przyjmując, że AI jako dziedzina zajmuje się badaniem zagadnienia, jak stworzyć sztuczną wersję prawdziwej inteligencji, jak mogłaby to sugerować sama nazwa. Ale będziemy przeciwni takiej interpretacji, częściowo z powodu braku jasności takiego postawienia sprawy. W końcu czy na pewno wiemy, czym jest inteligencja? Oczywiście McCarthy działał prawdopodobnie celowo, proponując rozmytą definicję na początku istnienia dziedziny, tak aby nie była ona przeszkodą dla naukowców. Jednak obecnie, definicja, którą uważamy za lepszą, brzmi następująco:

Sztuczna inteligencja to dziedzina wiedzy zajmująca się badaniem programów komputerowych, które w nietrywialny sposób używają wcześniej zdefiniowanych zasad, żeby tworzyć nowe niezdefiniowane wcześniej zasady, które pomagają w osiąganiu zadanych celów.

Z definicji tej wynika, że aby program został uznany za sztuczną inteligencję, musi w znaczącym stopniu wykazywać się zmianą swoich decyzji w zależności od napotkanych sytuacji. Wyobraźmy sobie np. grę w kamień-papier-nożyce. Tworząc program grający w tę grę, moglibyśmy zaprogramować zasadę, że jeśli przeciwnik używa ponadprzeciętnej ilości „kamieni" to zawsze gramy w odpowiedzi „papier", podobnie dla innych możliwości. Teraz, jeśli przeciwnik zacznie nadużywać którejś możliwości to program go za to ukara. Jest to jednak podejście proceduralne, wyliczające wszystkie możliwości. Alternatywą byłoby zastosowanie ogólnej zasady z kilkoma przypadkami. Możemy stworzyć zasadę: jeśli przeciwnik nadużywa $\boldsymbol{x}$ to graj $\boldsymbol{y}$ które pokonuje $\boldsymbol{x}$. Teraz wystarczy, że system podstawi odpowiednie wartości zgodnie z kolejnością: kamień, papier, nożyce. W ten sposób mamy uniwersalną zasadę z trzema przypdkami, a więc spełniamy definicję sztucznej inteligencji, jeśli uznamy to działanie za nietrywialne. Głównym problemem tej definicji jest właśnie pojawiające się tam słowo 'nietrywialne', które możemy rozumieć tylko intuicyjnie. W pewnym sensie jednak to, co będzie się kryć pod tym wyrazem, będzie związane z naszą wiedzą w tej dziedzinie.

1.4 Jakie tematy poruszymy w tym kursie

Podstawą zrozumienia tego kursu jest pewna znajomość matematyki. Jednak jest on tak pomyślany, aby osoba nieposiadająca części niezbędnej wiedzy, takiej jak np. podstawy różniczkowania, mogła zrozumieć jak najwięcej. Zaawansowane tematy nie będą zawarte w tym tekście i nie to było myślą autora. Raczej niż wprowadzać jak najwięcej tematów, autor jest przekonany, że lepiej jest posiąść głębokie zrozumienie podstaw niż mieć nikłe pojęcie o przeróżnych pomysłach, nie będąc w stanie ich odtworzyć. Autor chce zawrzeć w tym kursie matematyczną optymalizację, sieci neuronowe, drzewa poszukiwań oraz uczenie przez wzmacnianie. Dla osoby pierwszy raz stykającej się z tematem może się to wydać niedużą ilością materiału, ekspert może myśleć, że pokryjemy każdy temat źle. W umyśle autora wszystkie te obiekcje są ważne, ale tematy zostały wybrane ze względu na tworzenie jak najbardziej zawierającej się w sobie historii, która byłaby interesująca dla czytelnika i jednocześnie prezentowała najpotężniejsze istniejące techniki.

Rozdział 2

Optymalizacja matematyczna

Optymalizacja matematyczna jest dziedziną rozpoczynającą się od największych nazwisk w matematyce, m.in. Newton i Gauss odkryli pierwsze iteracyjne metody poruszania się w stronę optimum, a Fermat i Lagrange znaleźli sposoby wynikające z rachunku różniczkowego na znajdywanie tych maksimów i minimów. Innym ważnym nazwiskiem dla dziedziny jest Dantzig, który stworzył algorytm Simplex, myląc nierozwiązany problem z zadaniem domowym. Mimo iż optymalizacja ma długi rodowód, to nadal jest rosnącą dziedziną z wieloma osobliwościami. Obecnie wykorzystuje się w niej zaawansowany aparat matematyczny, który pozwala na tworzenie lepszych metod optymalizacyjnych. Rozwiązania z tej dziedziny są wykorzystywane w mechanice do projektowania budynków czy przedmiotów, w ekonomii do modelowania zachowań ludzi czy też w elektrotechnice. Widzimy więc, że dziedzina ta istniała przed powstaniem sztucznej inteligencji i posiada wiele zastosowań. My spojrzymy na nią w wymiarze, w którym pozwoli nam ona osiągnąć, to czego potrzebujemy do działania naszych systemów oraz pokażemy kluczowe pomysły. Należy jednak zaznaczyć, że większość ludzi pracujących w AI używa bibliotek, które zawierają znane, najlepsze wersje optymalizatorów. Jednak aby stosować opisane metody z sukcesem, kluczowym jest zrozumienie wewnętrznego działania tych technik.

Podstawowym pomysłem stojącym za użyciem metod optymalizacyjnych w AI jest to, że często mamy rozległy zasób możliwości, z którego nie potrafimy w jasny sposób wybrać najlepszej możliwości bez ekstensywnej eksploracji. Pomyślmy np. o wyborze dania w restauracji szybkiej obsługi. Chcemy wybrać jedzenie do 500 kcal, które będzie nam najbardziej smakowało. Moglibyśmy np. zjeść BigMac'a, który ma 495 kcal i wyczerpuje za jednym razem cały nasz limit, jednak równie dobrze możemy wybrać małe frytki, mały shake i sos jogurtowo-koperkowy co sprawi, że zmieścimy się w limicie tym razem z 485 kcal. Takich możliwości jest naprawdę dużo. Zamiast sosu moglibyśmy wziąć, chociażby kawę albo zamiast shake'a dużą colę. W takich przypadkach przydaje się znajomość technik optymalizacji. Pozwalają nam one na przetestowanie dużej liczby możliwości i wybranie takiej, która najbardziej nam odpowiada. Do tego problemu powrócimy później. Zazwyczaj jednak skupiamy się na sytuacjach ciągłych, w których możemy wybrać dowolną ilość danej rzeczy. To tak jakbyśmy mogli zamówić 1,2 frytek lub 0,56 sosu. Ponieważ zazwyczaj restauracje nie pozwalają nam na taką

ekstrawagancję, to skupimy się na trochę innym przykładzie. Pomyślmy o sytuacji, w której chcemy wybrać najlepsze miejsce na wybudowanie domu. Oczywiście w teorii możemy uszeregować wszystkie dostępne miejsca i wybrać najlepsze, ale może być takich możliwości nazbyt wiele. Moglibyśmy przecież postawić nasz dom nad morzem lub w górach, na obrzeżu miasta lub w wiosce, na płaskim lub nierównym terenie, w końcu na skale lub na piasku. Ponieważ nie jesteśmy w stanie sprawdzić wszystkich tych możliwości, ze względu na ich ilość, to z pomocą przychodzi nam optymalizacja matematyczna. Odpowiada ona na pytanie: jaki jest najlepszy sposób, aby badać tę przestrzeń możliwości? Nasza przestrzeń ma pewną korzystną właściwość, a mianowicie może być opisana matematycznie. Jeśli weźmiemy pod uwagę położenie domu na mapie, to otrzymujemy przestrzeń kartezjańską z zaznaczonymi na niej punktami, które odzwierciedlają możliwe położenie domu. Mamy więc dwie potrzebne rzeczy dla optymalizatora, a mianowicie przestrzeń oraz możliwe położenia do testowania. Brakuje nam jednak jeszcze jednego. Metoda, której użyjemy, musi wiedzieć, jak bardzo cenimy dane położenie, aby być w stanie dać nam coraz to lepsze rezultaty. Tak więc załóżmy, że dla każdego pojawiającego się położenia będziemy wpisywać wartość ręcznie. Z takim przygotowaniem możemy użyć metod optymalizacyjnych. Jak to będzie działać? Jeśli poruszymy się, powiedzmy w pierwszym kroku w stronę morza, a my wolimy góry, to przypiszemy nowemu położeniu niższą wartość. Optymalizator sam 'zorientuje się', że należy kierować się w przeciwnym kierunku i w następnych krokach zmieni sposób poruszania się.

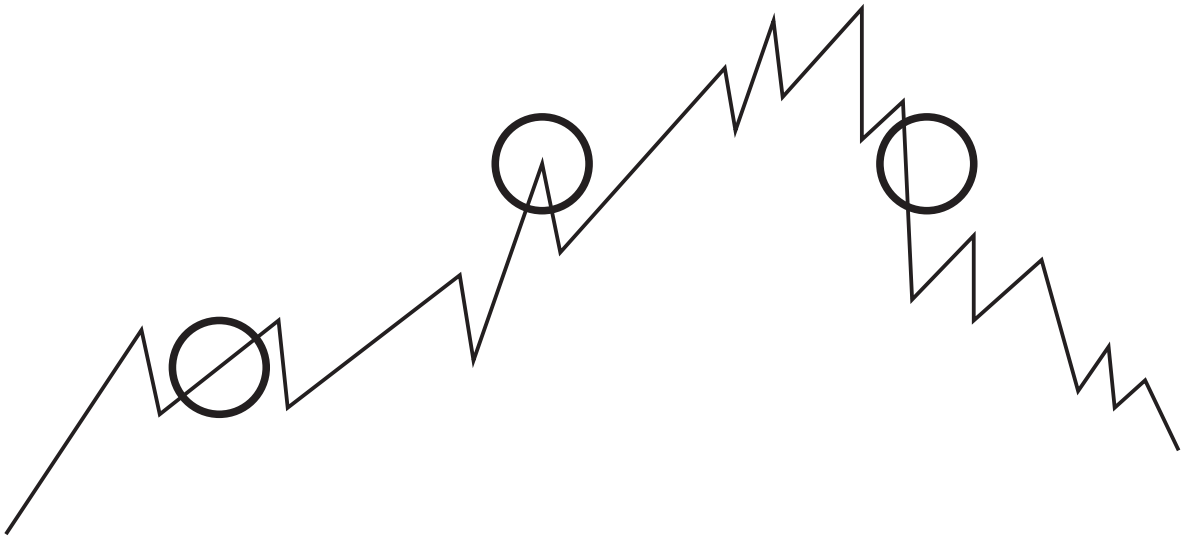
2.1 Metoda prób i błędów

Spróbujemy teraz sformalizować, to co opisaliśmy w poprzednim podrozdziale. Możesz nadal myśleć o wyborze najkorzystniejszego położenia na dom, nie będzie to kolidować z formalną definicją. Problem jest określony następująco: Mamy zmienną $\boldsymbol{y}$ którą chcemy optymalizować, to znaczy sprawić, aby była jak największa. Zmienna $\boldsymbol{y}$ zależy od zmiennej $\boldsymbol{x}$ w jakiś nieznany nam sposób. Jednak możemy wpływać na $\boldsymbol{x}$ poprzez podejmowanie akcji. Wyobraźmy sobie, że możemy go zmieniać, tak jak zmienialiśmy położenie domu.

$$max(y = f(x)) \tag{2.1}$$

Gdzie $\boldsymbol{x}$ jest zmienną, na którą mamy wpływ, $\boldsymbol{f}$ jest transformacją, której zazwyczaj nieznamy, **max(y)** mówi nam, że $\boldsymbol{y}$ jest celem, inaczej mówiąc wielkością, którą chcemy maksymalizować.

Ryc. 2.1: Metoda prób i błędów



Metoda prób i błędów proponuje nam, abyśmy zgadli wartość $\boldsymbol{x}$ i zapisali związaną z nią wartość $\boldsymbol{y}$. Następnie wielokrotnie powtarzamy fazę zgadywania i zapisu. Po jakimś czasie takiego zgadywania wybieramy $\boldsymbol{x}$ które posiadało związaną z nią najwyższą wartość $\boldsymbol{y}$.

Jest to najprostsza możliwa metoda optymalizacji. Żeby ją stosować, wystarczy utrzymywać w pamięci najlepsze rozwiązanie, na które napotkaliśmy do tej pory i zgadywać kolejne rozwiązania, które mogą być lepsze, czyli najlepiej takie, których poprzednio nie sprawdzaliśmy. Tą metodą możemy sprawdzić każdą możliwą sytuację, jeśli tylko przestrzeń akcji, którą badamy, jest skończona. Mapa, na której szukamy miejsca na dom, powinna być ograniczona np. sąsiadującymi krajami i posiadać ograniczenie małą wielkość kroku. W takiej sytuacji, jeśli damy tej metodzie wystarczająco dużo czasu to doprowadzi nas ona do najlepszego istniejącego rozwiązania. Jest to niewątpliwie silna strona tego podejścia, która pokazuje generalność tej metody. Ta prosta metoda ma jednak również pewne problemy, jak nieskończone przestrzenie powodujące, że wyszukiwanie potrzebuje nieskończonego czasu, aby znaleźć najlepsze rozwiązanie. Także z reguły, czas, który zajmuje ta metoda, w porównaniu do innych jest daleki od najlepszych. Co powoduje taką nieefektywność? Metoda prób i błędów nie posiada żadnej metryki, która pokazywałaby nam, jak bardzo poprawia się nasz wynik w miarę szukania. Można to porównać do szukania przycisku do zapalania światła po omacku. Nie jest to łatwe zadanie, chociaż po pewnym czasie się nam to uda. Sytuację z wyłącznikiem światła poprawia na dodatek fakt, że zazwyczaj wiemy, mniej więcej, w jakim kierunku się on znajduje. Jednak jeśli trafilibyśmy do zupełnie nowego pokoju, to problem ten byłby o wiele trudniejszy. Z takimi trudnościami boryka się metoda prób i błędów. Aby poprawić szybkość szukania, potrzebujemy jakiejś miary, która będzie nam mówić, czy poprawiamy nasz wynik, czy nie. Tak samo gra w szukanie wyłącznika staje się dużo łatwiejsza, kiedy ktoś mówi nam ‘ciepło’ lub ’zimno’ wraz z poruszaniem się po pokoju. Z podobnego rodzaju informacji powinien korzystać nasz algorytm.

2.2 Wspinaczka górska

Pomyślmy, jak moglibyśmy prowadzić nasze poszukiwania, tak aby nie sprawdzać tak wielu złych rozwiązań. Jakiego rodzaju informacji moglibyśmy użyć? Jedną z metod, która mogłaby nam pomóc, jest lepszy wybór punktów do sprawdzenia. Jeśli przestrzeń jest wystarczająco ciągła, to znaczy punkty leżące nieopodal siebie mają podobne właściwości, to moglibyśmy użyć wiedzy o punktach leżących blisko, żeby przybliżać wartość szukanego punktu. Zupełnie jak w grze ciepło-zimno, jeśli wiemy, że jesteśmy w miejscu, w którym niedaleko było ‘zimno’, to prawdopodobnie tu gdzie

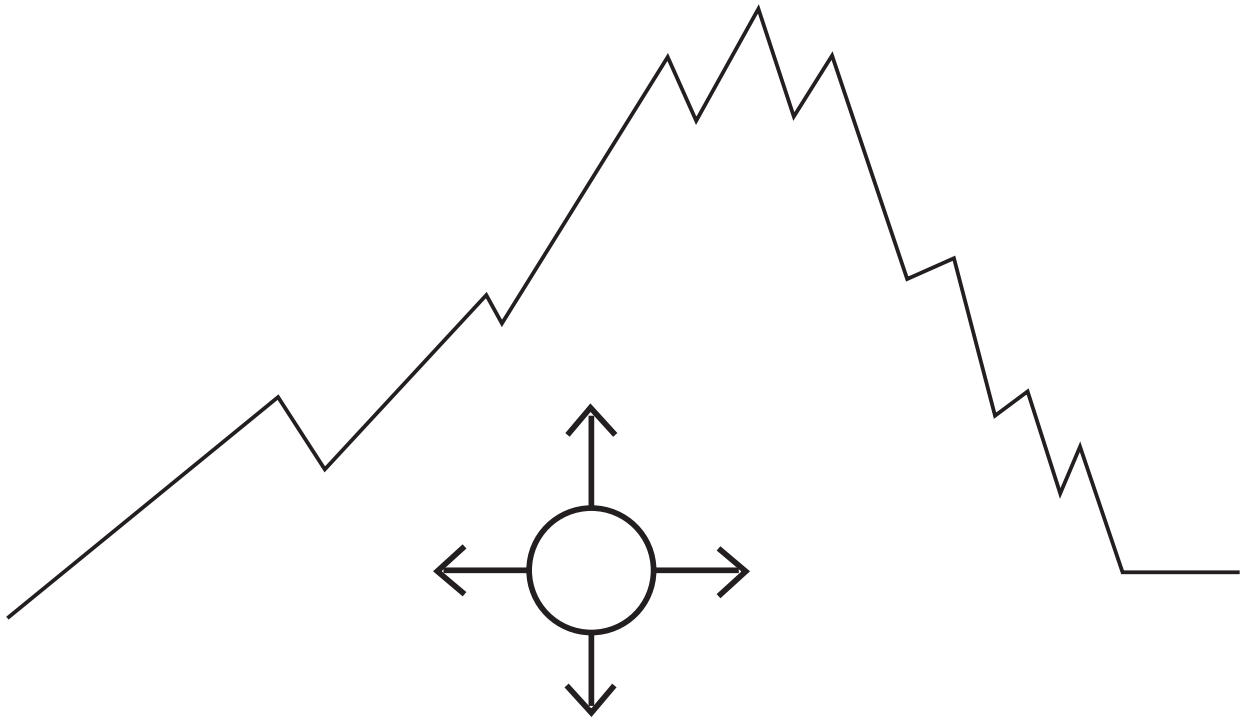
jesteśmy, też będzie ‘zimno’. Tak więc możemy skupić nasze poszukiwania jedynie na punktach, w których jest ‘ciepło’, czy mówiąc inaczej na punktach, które dają nam dobrą wartość numeryczną funkcji wartości. Ponieważ chcemy prowadzić nasze poszukiwania na dowolnie dużych przestrzeniach, weźmiemy pod uwagę tylko poszukiwanie lokale, czyli takie, które skupia się na wyszukiwaniu optimów lokalnych. Zazwyczaj taki rodzaj poszukiwania ma w danym momencie tylko jednego kandydata na rozwiązanie.

Tu możemy wspomnieć, że istnieją też tak zwane poszukiwania globalne takie, jak np. algorytm mrówkowy, który bierze swoją nazwę od symulowania zachowania owadów, jakimi są mrówki. Tworzy on wiele ‘mrówek’, każda ze swoim osobnym zachowaniem. Akcje poszczególnych mrówek wpływają na siebie tak, że zachowanie całego roju jest bardziej inteligentne niż poszczególnych osobników. Algorytm ten działa na zasadzie zaobserwowanej w świece zwierząt. Mrówki poruszają się po losowych ścieżkach, chyba że znajdą pożywienie. Wtedy wracają do mrowiska, pozostawiając za sobą ślad z feromonów. Inne mrówki napotykając na ślad z feromonów, zaczynają nim podążać, w efekcie samemu wydzielając feromony. W ten sposób uczęszczana ścieżka jest wzmacniana. Kiedy jednak jedzenie się kończy, wydzielanie feromonów zostaje zaprzestane, a ścieżka naturalnie zanika. Podobnego sposobu możemy użyć do znajdywania najlepszych ścieżek na grafie. To nazywane jest poszukiwaniem globalnym, ponieważ próbuje znaleźć globalnie najlepsze rozwiązanie, czyli takie, które jest najlepsze dla całej przestrzeni. Takie poszukiwanie często utrzymuje w pamięci wiele punktów związanych z danym problemem i próbuje znaleźć rozwiązanie, biorąc pod uwagę je wszystkie. W przeciwieństwie do poszukiwania globalnego poszukiwanie lokalne szuka najlepszego rozwiązania tylko w pewnej okolicy. Zazwyczaj utrzymuje też w pamięci informacje o tylko jednym punkcie. Mogłoby się nam wydawać, że poszukiwanie globalne będzie nam bardziej przydatne niż lokalne, jednak najpowszechniej wykorzystywanymi algorytmami w dziedzinie sztucznej inteligencji są algorytmy wyszukiwania lokalnego. To podejście ma jedną ważną korzyść, a mianowicie możliwość użycia informacji o gradiencie. Na razie nie musimy się jednak o to martwić.

O takim sposobie lokalnego poszukiwania możemy myśleć jak o wspinaczce górskiej. Wyobraźmy sobie, że znajdujemy się u podnóża góry, na której szczyt chcemy się wspiąć. Mamy jednak jeden problem. Nie widzimy wierzchołka, gdyż cała góra została pokryta mgłą. Teraz, aby dostać się na szczyt, możemy podjąć następującą metodę działania. Patrzymy się na punkty w widocznej okolicy i wybieramy ten, który znajduje się najwyżej. W jego kierunku się udajemy. Powtarzamy ten proces, dopóki wszystkie punkty znajdujące się obok nas są niżej położone i wtedy stwierdzamy, że znaleźliśmy się na szczycie i kończymy poszukiwania. Oczywiście nie jest powiedziane, że w efekcie działania tej metody znajdziemy się na oczekiwanym szczycie. Może się

np. okazać, że wejdziemy na górę znajdująca się obok. Może też się stać tak, że nie wejdziemy na żaden szczyt, tylko utkniemy na pagórku. Ta właściwość tego poszukiwania nazywana jest problemem maksimum lokalnego. Kiedyś uważano, że maksima lokalne są główną przeszkodą w osiąganiu najlepszego rozwiązania. Obecnie poglądy na ten temat uległy zmianie i za najgorsze są uważane płaskie przestrzenie, jednak nie ulega wątpliwości, że możliwym jest utknięcie w maksimum lokalnym. Aby temu zapobiegać, wymyślono metodę losowego resetu (ang. random-restart), która wykonuje wspinaczkę górską wielokrotnie za każdym razem z losowo wybranego stanu początkowego. Ma to służyć omijaniu minimów lokalnych. Jednak wybiegliśmy do przodu, powiedzmy najpierw, czym w ogóle jest algorytm wspinaczki górskiej.

Ryc. 2.2: Algorytm wspinaczki górskiej



Algorytm **wspinaczki górskiej** (ang. hill climbing) utrzymuje w pamięci jedno obecnie najlepsze rozwiązanie, sprawdza punkty wokół i porusza się w kierunku największej poprawy wartości, tak długo, jak jakakolwiek może być otrzymana. W tym algorytmie wybór punktów do sprawdzenia odbywa się w pewien specyficzny sposób. Wyobraźmy sobie, że zamiast wybierać najwyższy punkt, sprawdzamy cztery kierunki świata i udajemy się w kierunku, w którym poprawa jest największa. Analogiczne działanie ma algorytm wspinaczki górskiej. W $\boldsymbol{n}$ wymiarach przed zmianą pozycji powinniśmy sprawdzić 2 * $\boldsymbol{n}$ ortogonalnych kierunków. To znaczy kierunków, które znajdują się wobec siebie pod kątem prostym w $\boldsymbol{n}$ wymiarach.

Ten problem tego podejścia wydaje się oczywisty, wraz ze wzrostem ilości wymiarów ilość możliwości, które musimy sprawdzić, rośnie w ogromny sposób. Również niekoniecznie poruszamy się w najlepszym kierunku, ponieważ może on leżeć pomiędzy kierunkami, które sprawdzaliśmy. Wybieramy tylko jeden spośród 2 * $\boldsymbol{n}$ kierunków, który jest najbliżej optymalnego kierunku wchodzenia w górę. Nie użyliśmy też żadnych informacji na temat tak zwanego gradientu.

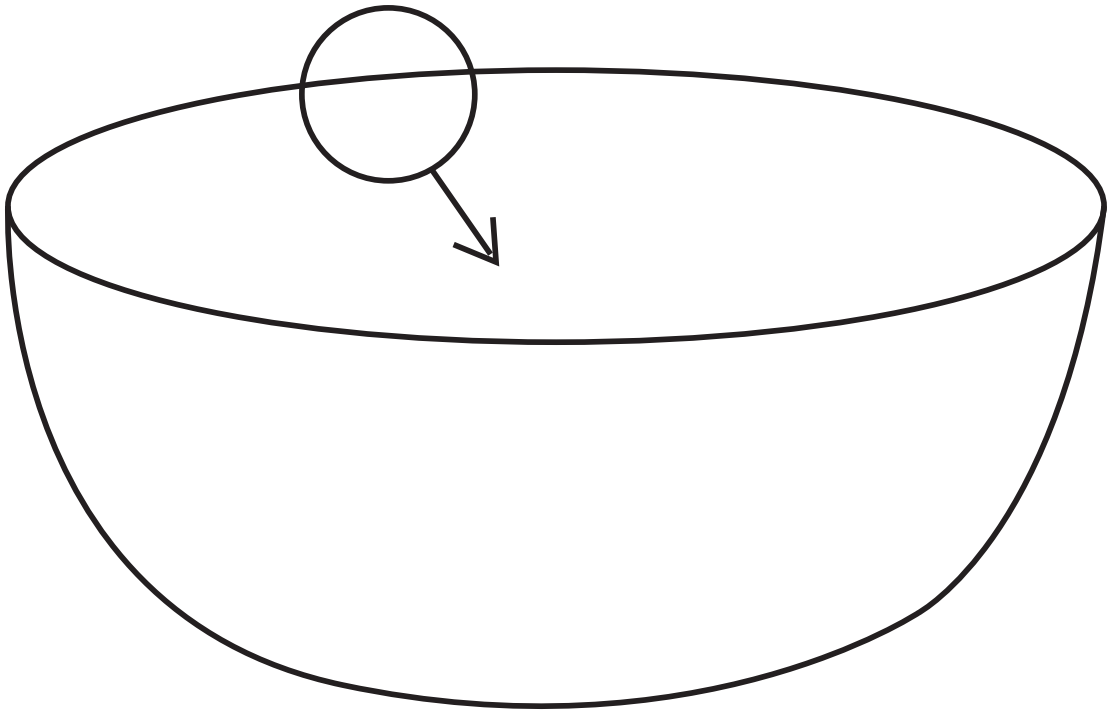
2.3 Metoda gradientu prostego

Rozmawialiśmy o poruszaniu się w górę zbocza, zwiększając jakąś wartość, ale ludzie ze środowiska optymalizacyjnego z niezbyt jasnych powodów (prawdopodobnie chodzi o jasność notacji) zazwyczaj mówią o poruszaniu się w dół, zmniejszając jakąś wartość. Te pomysły są identyczne poza zmianą znaku. Jeśli konceptualnie odwrócimy przestrzeń na głowie, to możemy zamienić wspinanie się ze schodzeniem w dół i na odwrót.

Mówiliśmy o problemach, jakie są związane z algorytmem wspinaczki górskiej. Głównym problemem jest jednak to, że pomija on pewien rodzaj informacji. Czy możesz wskazać jakiś rodzaj takich informacji, które pozostają nieużyte, a mogłyby nam pomóc? Jednym z problemów jest to, że nie patrzyliśmy się na różnicę między wartością naszej funkcji wartości w poprzednim punkcie a obecnym, żeby znaleźć lepszy kierunek schodzenia. Jak mogłaby nam ta różnica pomóc? Wyobraźmy sobie, że jest ona duża. Oznaczałoby to, że poruszamy się w dobrym kierunku. Jeśli jednak ta różnica jest niewielka, to może powinniśmy się poruszać w innym kierunku, który mógłby nam potencjalnie dać większy zysk funkcji wartości. Teraz weźmy pod uwagę kilka kolejnych kroków i odpowiadające im zmiany funkcji wartości. Chcemy wymyślić sposób wykorzystania tych informacji do wybrania kolejnego kierunku, który będzie potencjalnie najlepszy. Takie połączenie jednych informacji z innymi jest nazywane korelacją. Jeśli mielibyśmy metodę korelacji kierunków, w których poruszaliśmy się z funkcją

wartości, to moglibyśmy powiedzieć że kiedy wybierzemy kierunek o historycznie największym spadku to jest to według naszej wiedzy najlepszy kierunek. Taką korelację można znaleźć w bardzo prosty sposób. Wystarczy podzielić zmianę funkcji wartości poprzez wartość kroku w danym kierunku, a później wszystko dodać do odpowiednich kierunków, żeby otrzymać pewne przybliżenie zmiany w każdym kierunku. Następnie wybieramy kierunek o największej historycznej poprawie funkcji wartości. Ponieważ jednak krzywizna przestrzeni będzie się zmieniać, to taka procedura będzie jedynie pewnym przybliżeniem. Najprościej jest jednak założyć, że zmiana w danym kierunku pozostanie taka sama i zachowywać się jak gdyby było to prawdą. Metoda ta może być przydatna nawet w życiu codziennym. Wyobraźmy sobie, że robimy w ciągu dnia różne rzeczy i zapisujemy, jak dobrze czujemy się danego dnia. Później, aby uzyskać ranking rzeczy, które będziemy robić, moglibyśmy rozdzielić to jak dobrze się czuliśmy w proporcjach do tego jak dużo wykonywaliśmy danej czynności dla każdego dnia i dodać wyniki z różnych dni. To dałoby nam numeryczne wartości określające pożądanie danych zachowań, dzięki którym moglibyśmy je uszeregować.

Ryc. 2.3: Metoda gradientu prostego



Spójrzmy teraz na bardziej matematyczny przykład w 2-dim (dwóch wymiarach): Zaczynamy w punkcie $\boldsymbol{p_0 = (x_0, y_0)}$ i poruszamy się o wektor **(1, 2)**. Kończymy w punkcie $\boldsymbol{p_1} = (\boldsymbol{x_0} + 1, \boldsymbol{y_0} + 2)$ albo w innej metodzie zapisu w punkcie $\boldsymbol{p_1} = (\boldsymbol{x_1}, \boldsymbol{y_1})$. Powiedzmy teraz, że poprawiliśmy naszą funkcję wartości z $\mathbf{y} = \boldsymbol{y_0}$ do $\mathbf{y} = \boldsymbol{y_1}$.

Metoda gradientu prostego mówi nam, że nasz następny kierunek, w którym będziemy się poruszać, powinien być proporcjonalny do poprawy naszej funkcji wartości (a więc w przykładzie do $\boldsymbol{\Delta y} = \mathbf{y1} - \mathbf{y0}$) i także odwrotnie proporcjonalny do kierunku, w którym się poruszaliśmy (więc wektora **(1, 2)**). W przykładzie następny wektor ruchu będzie równy $\boldsymbol{(\Delta y}$ **/ 1,** $\boldsymbol{\Delta y}$ **/ 2)** a ogólnie:

$$gradient = \Delta y / \Delta x \tag{2.2}$$

Gdzie **gradient** oznacza, ogólnie mówiąc kierunek największego spadku, $\boldsymbol{\Delta y}$ to zmiana funkcji wartości, a $\boldsymbol{\Delta x}$ oznacza przesunięcie, o które poprzednio się poruszaliśmy.

Co prawda pokazaliśmy przykład działania na gradiencie empirycznym otrzymanym z wykonania kroku optymalizacyjnego, lecz lepszą metodą osiągnięcia tego samego celu jest znalezienie pochodnej funkcji, którą optymalizujemy. Pochodna jest tu sposobem na obliczenie równania (2.2) dla $\boldsymbol{\Delta y}$ oraz $\boldsymbol{\Delta x}$, dla których $\boldsymbol{\Delta x}$ dąży do zera, a więc intuicyjnie jest to najlepszy kierunek poruszania się dla nieskończenie małego kroku.

$$gradient = f'(x) \tag{2.3}$$

Gdzie $\boldsymbol{f'(x)}$ oznacza pochodną pierwszego stopnia z funkcji $\boldsymbol{f}$ w punkcie $\boldsymbol{x}$. Powinniśmy korzystać z tego równania zamiast (2.2) zawsze kiedy umiemy wyznaczyć pochodną optymalizowanej funkcji.

Aby otrzymać nowy punkt do sprawdzenia, musimy wziąć informacje o poprzednim punkcie i informacje o gradiencie i przetworzyć je w taki sposób, aby otrzymać nowy punkt. Najprostszym sposobem na to jest odjąć gradient od starego punktu. Pamiętamy przy tym, że szukamy największego spadku. Jeśli to zrobimy, to otrzymamy tak zwaną formułę rekursywną (co znaczy, że stosujemy ją wielokrotnie na wyniku tej samej formuły) dla znajdywania lepszych punktów:

$$p_{n+1} = p_n - gradient \tag{2.4}$$

Gdzie $\boldsymbol{p_{n+1}}$ jest nowym punktem, z którego będziemy kontynuować poszukiwania, $\boldsymbol{p_n}$ jest starym punktem, a **gradient** jest kierunkiem największego spadku, w którym się poruszamy.

Największym pomysłem, którego tutaj używamy, jest fakt że możemy użyć przeszłej informacji na temat zakrzywienia przestrzeni którą badamy żeby prowadzić nasze wyszukiwania w najlepszym kierunku. Korzystamy z tego, iż niedalekie punkty są często podobne do siebie, np. jeśli jesteśmy w płaskim regionie, to spodziewamy się że jeśli przesuniemy się kawałek dalej to teren nadal będzie płaski. Ten algorytm wraz z pewnymi dodatkami, część, z których zostanie opisana w kolejnych rozdziałach stanowi podstawowy i najczęściej używany algorytm dla optymalizacji w okolicznościach spotykanych w dziedzinie sztucznej inteligencji.

2.4 Plus wielkość kroku

Powiedzieliśmy trochę o wyborze kierunku ruchu, ale nie skupialiśmy się do tej pory na metodach wyboru wielkości kroku. Czytając poprzedni podrozdział, mogłeś myśleć czytelniku, że gradient jest pewnym idealnym krokiem, o który należy się poruszyć niezależnie od sytuacji. Jest prawdą, że gradient podaje nam kierunek największego spadku, lecz nic nie powiedzieliśmy o tym, czy wskazuje on na optymalną odległość. Tak nie będzie. Metody gradientu są więc prawie zawsze, używane wraz z rozwiązaniami, które wybierają wielkość kroku, który mówi nam, jak daleko powinniśmy się przesunąć w kierunku gradientu. Potrzebujemy więc jakiegoś sposobu na określenie wielkości potrzebnego kroku. Najprostszym rozwiązaniem jest dodać do równań pewną zmienną, która będzie określać wielkość kroku. Możemy ją nawet nazwać wielkością kroku. Poprzednie równania zmienią się w następujący sposób:

$$gradient = dy/dx \tag{2.5}$$

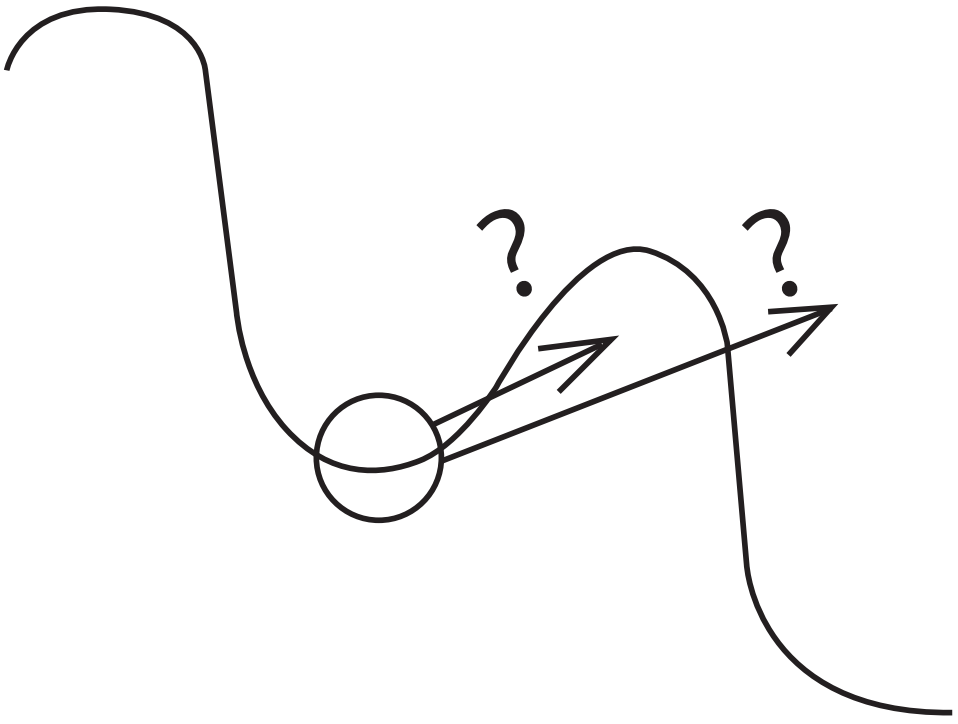
$$p_{n+1} = p_n - step\_size * gradient \tag{2.6}$$

Gdzie równanie (2.4) jest identyczne z równaniem (2.3), a w równaniu (2.5) które jest analogiczne do równania (2.4), pojawia się jednak nowa wartość zwana **step_size** czyli wielkość kroku, ta wielkość skalarna będzie nam mówić, jak daleko przesuniemy się za każdym razem.

Należy zaznaczyć, że dodanie wielkości kroku nie zamyka problemu, konieczności odpowiedniego jego wybrania. Teraz możesz się zastanowić, jak powinniśmy dokonać wyboru wielkości kroku. Czy istnieje optymalny sposób na dokonanie tego? Odpowiedź brzmi: Nie, nie ma jednej najlepszej metody wyboru kroku dla każdego problemu. Ta właściwość jest spowodowana faktem, że problemy, które możemy chcieć próbować rozwiązać, mogą być od siebie bardzo różne. Wyobraźmy sobie, że możemy chcieć wspinać się na Babią Górę lub na Rysy, dla każdej z tych gór wielkość kroku, który

będziemy robić, będzie inny, dopasowany do nachylenia zbocza oraz ilości przeszkód. Od podobnych czynników będzie zależeć optymalna wielkość kroku w systemach AI. Taka sytuacja jest częsta w tej dziedzinie i jeśli będziesz kontynuował poszerzanie swojej wiedzy o niej, to napotkasz na tę właściwość wielokrotnie. Jedną z rzeczy, która odróżnia fizykę i sztuczną inteligencję jest fakt, że w AI nie ma znanych stałych jak np. **step_size** = 0.234... czy coś podobnego. To także oznacza, że ktoś próbujący wytrenować model, na przykład, taki o jakim będzie mowa w rozdziale o sieciach neuronowych, będzie zmieniać taką zmienną, żeby zobaczyć czy poprawia to działanie systemu. Trzeba w tym miejscu zaznaczyć, że takie nudne zmienianie pewnych wartości jest znaczącą częścią pracy kogoś, kto zajmuje się tym zagadnieniem. Ostatecznie, kiedy pewna metoda zostanie opracowana to to, co pozostaje praktyką to dostosowanie jej do swojego konkretnego przypadku, na co zazwyczaj składa się właśnie precyzyjny dobór wielu parametrów. Mimo iż nie ma jednego najlepszego kroku, to istnieją sposoby, które mogą nam pomóc w wyborze takiej stałej. Jak myślisz, w jakim przedziale powinna się znaleźć wielkość kroku? Zwyczajna odpowiedź to gdzieś pomiędzy 0,2 a 0,000001 co jest jednak dosyć dużym przedziałem. Czy możesz pomyśleć o dodatkowej technice, która mogłaby nam pomóc w wyborze kroku? Jednym z istniejących rozwiązań jest tzw. **planowane zmniejszanie kroku** (ang. learning rate scheduling) Istotą tego podejścia jest zmniejszanie kroku wraz z upływem czasu. Powoduje to dużą eksplorację na początku uczenia i coraz dokładniejsze przeczesywanie najlepszych części przestrzeni pod koniec nauki. Wielkość kroku ma też inną nazwę, a mianowicie **wskaźnik uczenia** (ang. learning rate), ponieważ wpływa na tempo nauki. Zatrzymajmy się nad tym na chwilę, ponieważ jest to bardzo ważna zależność. Kiedy dokonujemy dużych kroków, to poruszamy się przez przestrzeń szybciej niż gdybyśmy wybrali mniejszy krok, możemy więc powiedzieć, że szybko się uczymy. Nie odbywa się to jednak bez kosztu. Robiąc szybkie postępy, możemy nie zauważyć drobnych niedokładności, jakie mają miejsce przy okazji. Dlatego, gdy postęp zaczyna maleć najlepiej zmniejszyć wskaźnik uczenia i dokonać koniecznych poprawek na mniejszą skalę. W naszym przypadku duży wskaźnik uczenia da nam szybką poprawę, ale może przeskoczyć ponad pewnymi dobrymi miejscami. Za to mały wskaźnik uczenia da nam wolniejszą poprawę, ale sprawi, że nie ominiemy żadnych dobrych regionów. Jedną z metod połączenia tych dwóch korzyści jest planowane zmniejszanie kroku, gdzie wraz z upływem czasu lub po zobaczeniu pewnej ilości przykładów wskaźnik uczenia zostaje pomniejszony. Możemy w pewien sposób myśleć o tym procesie jak o zmniejszaniu temperatury. Przy dużej temperaturze cząsteczka porusza się bardzo gwałtownie, a wraz ze zmniejszaniem temperatury cząstka ogranicza się do dołków energetycznych, czyli regionów gdzie jej energia jest zminimalizowana. Podobny proces możemy zaobserwować w hutnictwie gdzie wyżarzanie, polegające na podgrzaniu i następnym schłodzeniu materiału jest wykorzystywane w celu sprawienia, aby materiał był bardziej podatny dalszej obróbce.

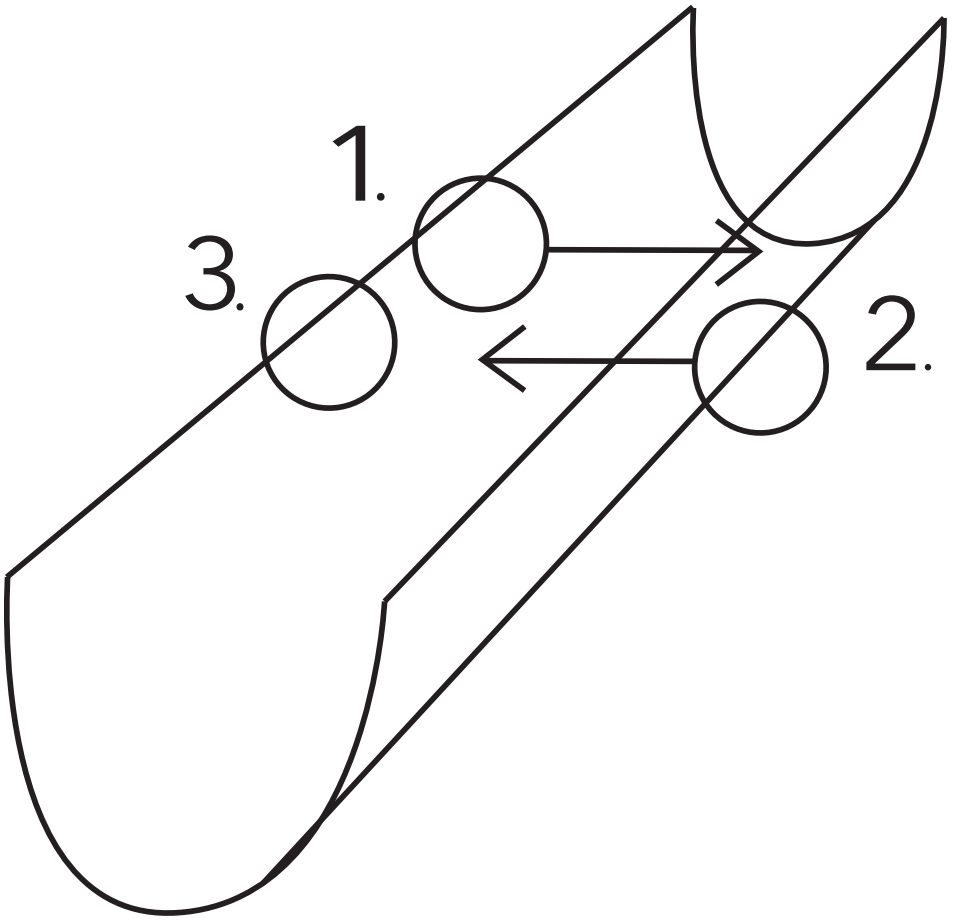
Ryc. 2.4: Wybór wielkości kroku



2.5 Plus momentum

Problemem metod gradientu prostego jest to, że często nie jest wydajna. Żeby zobaczyć dlaczego, zastanówmy się nad następującym przykładem. Znajdujemy się w wąskim lekko pochylonym w jedną stronę wąwozie, z dwóch stron mamy strome ściany. Żeby zrobić postęp w takim miejscu, trzeba iść dnem wąwozu, ale wyobraźmy sobie, że znajdujemy się na ścianie takiego wąwozu. Teraz logicznym wydaje się, że trzeba zejść na dno i iść dnem takiego wąwozu. Zobaczmy jednak, jak zachowuje się algorytm gradientu prostego w takiej sytuacji. Jak myślisz, co będzie wynikiem jego działania? Odpowiedź brzmi: algorytm gradientu prostego będzie się poruszał w kierunku przeciwnym do kierunku biegu wąwozu, ponieważ jeśli jest on odpowiednio wąski, to będziemy przeskakiwać od razu na przeciwną ścianę. Kiedy już znajdziemy się na przeciwnej ścianie to podobnie, w następnej iteracji, przeskoczymy z powrotem na ścianę, z której zaczęliśmy. Żeby zobaczyć, dlaczego tak się dzieje, pomyśl, w jakim kierunku porusza się algorytm gradientu prostego. Porusza się on zawsze w kierunku największego gradientu, inaczej mówiąc w kierunku największego spadku. Dla ściany wąwozu największy spadek jest w dół wąwozu, a nie w kierunku biegu wąwozu. Niestety nie będziemy się poruszać w kierunku prawdziwej poprawy wartości, a raczej tymczasowej, która zostanie wymazana poprzez następujący za chwilę niechybny powrót na ścianę, z której zaczynaliśmy. Nasza droga będzie przypominać powolne zygzakowanie w poprawnym kierunku. Pochyłość wąwozu sprawi co prawda, że będziemy się poruszali w dobrą stronę, ale będzie to bardzo powolny postęp. Żeby zapobiegać takiemu niekorzystnemu zjawisku wprowadzono pęd (ang. momentum).

Ryc. 2.5: Zachowanie optymalizatora bez dodatku pędu



W fizyce pęd jest zdefiniowany jako:

$$R = m * v \tag{2.7}$$

Gdzie $\boldsymbol{R}$ oznacza pęd, $\boldsymbol{m}$ masę a $\boldsymbol{v}$ prędkość.

Siła zdefiniowana jest jako:

$$F = d(mv)/dt \tag{2.8}$$

A więc jest pochodną pędu po czasie.

Teraz wiedząc, że siła jest zmianą pędu, wyobraźmy sobie, że czas podzielony jest na małe części. Wtedy, żeby dostać pęd w n-tym kroku, należałoby obliczyć wartość następującego rekursywnego równania:

$$R_{n+1} = R_n + F \tag{2.9}$$

Ponieważ $\boldsymbol{F}$ jest zmianą w $\boldsymbol{R}$.

Jest jeszcze jeden mały szczegół, którego nie bierzemy pod uwagę. W prawdziwym świecie obserwujemy tarcie, które sprawia, że każdy ruch traci swój pęd w miarę upływu czasu. To tarcie nazwiemy $\boldsymbol{\alpha}$ i będzie ono opisywać procent pędu, który nie jest wytracany w jednym kroku czasu. Równanie zmienia się następująco:

$$R_{n+1} = \alpha * R_n + F \tag{2.10}$$

To jest równanie na pęd w metodzie gradientu, jeśli tylko podstawimy odpowiednie wartości za $\boldsymbol{R}$ i $\boldsymbol{F}$. Przypomnijmy sobie teraz równanie służące do znajdywania lepszych punktów (2.5):

$$p_{n+1} = p_n - step\_size * gradient \tag{2.11}$$

Teraz, zamiast używać zmiany **step_size** * **gradient** zastąpmy ją pędem.

$$p_{n+1} = p_n + R_{n+1} \tag{2.12}$$

Następnie rozwińmy $\boldsymbol{R_{n+1}}$ przy pomocy równania (2.9):

$$p_{n+1} = p_n + \alpha * R_n + F \tag{2.13}$$

Jak mówiliśmy, musimy podstawić odpowiednie wartości. Pęd $\boldsymbol{R_n}$ zastąpimy zmienną, która będzie oznaczać zmianę wartości punktu $\boldsymbol{p_n}$ dla poprzedniej iteracji. Jednak ta zmiana jest zależna od drugiej części równania ($\boldsymbol{\alpha * R_n + F}$), które opisuje tę zmianę, więc jak możemy zauważyć pęd $\boldsymbol{R_{n+1}}$ będzie zależny od siebie samego w

poprzedniej iteracji. Tak więc musimy utrzymywać w pamięci drugą część równania, aby użyć go ponownie w następnej. Za to $\boldsymbol{F}$ które w fizyce jest zmianą w pędzie, zastąpimy zmianą zmiany wartości punktu czy inaczej zmianą pędu dla punktu. $\boldsymbol{F}$ będzie więc równe gradientowi, który będzie określał tę zmianę tak, aby pęd zbliżał się do optymalnego pędu. Równanie (2.9) ze zmienionymi zmiennymi prezentuje się następująco:

$$\Delta p_{n+1} = \alpha * \Delta p_n - gradient \tag{2.14}$$

A równanie (2.12) otrzyma ostateczną formę:

$$p_{n+1} = p_n - gradient + \alpha * \Delta p_n \tag{2.15}$$

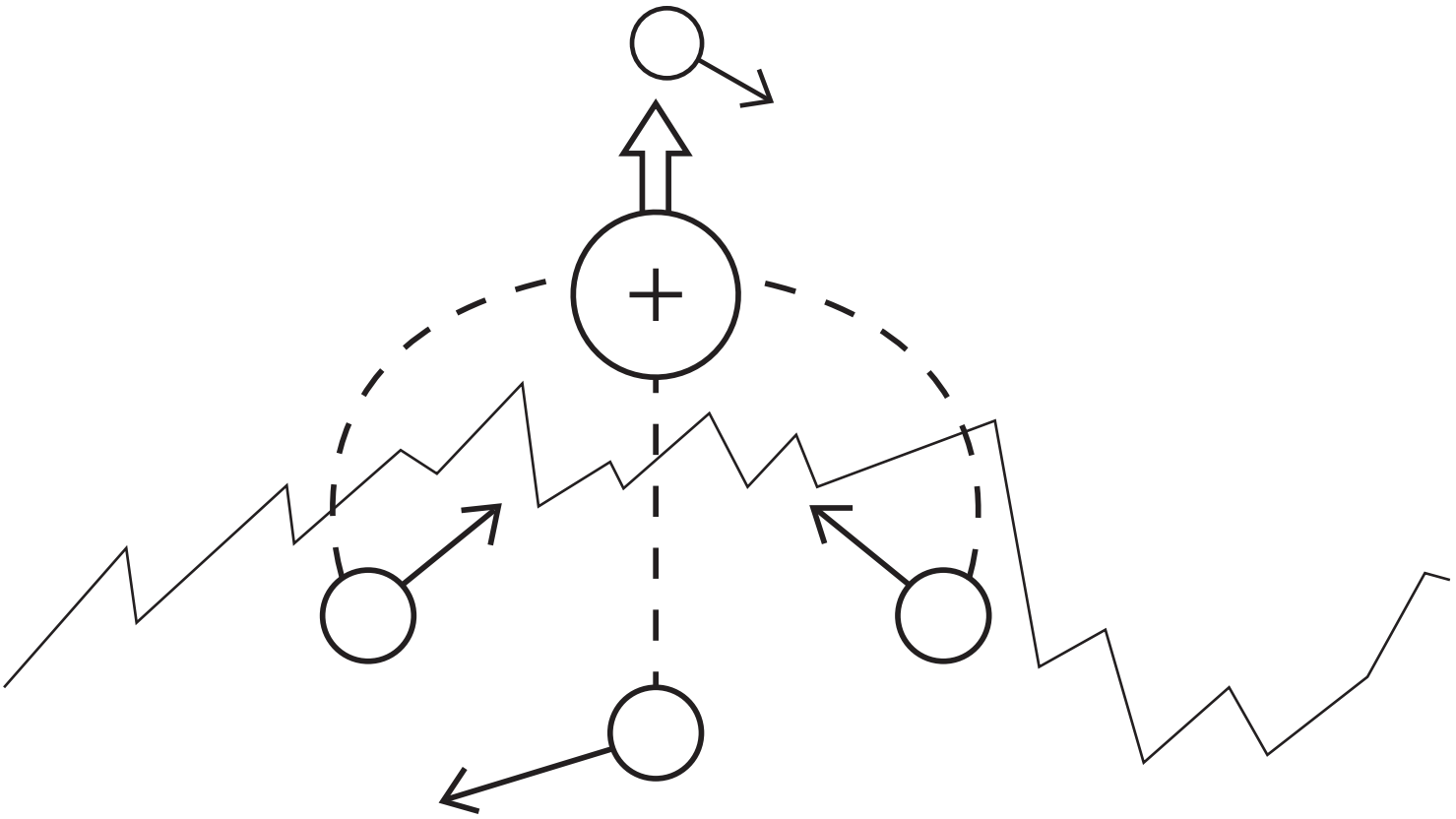
Gdzie $\boldsymbol{p_n}$ jest wartością wyszukiwania w n-tej iteracji, **gradient** jest gradientem, $\boldsymbol{\alpha}$ jest parametrem 'tarcia', a $\boldsymbol{\Delta p_n}$ określa zmianę w $\boldsymbol{p_n}$ dla poprzedniej iteracji. Tu znowu $\boldsymbol{\alpha}$ jest stałą, którą możemy zmieniać, bo nie ma dla niej jednej najlepszej wartości dla każdego problemu. Wszystko co następuje po minusie jest identyczne jak prawa strona równania (2.13) określającego nasz pęd.

Zauważmy jeszcze jedną rzecz, która mogła cię zaniepokoić. Przed gradientem postawiliśmy znak minus. Wynika to z tego, że gradient określa kierunek największego wzrostu funkcji. Ponieważ my chcemy ją minimalizować, to powinniśmy się udać w przeciwnym kierunku. Jeśli jeszcze kiedyś zaniepokoisz się znakiem, to pomyśl czy nie wynika to właśnie z tej zależności. Wracając do metafory wąwozu, równanie (2.13) sprawia, że będąc w wąskim przesmyku, będziemy dodawać poprzedni pęd do obecnego. Będą się one nawzajem niwelować, ponieważ będą skierowane w przeciwnych kierunkach, a to, co zostanie, będzie kierowało nas w dół wąwozu, wygładzając naszą podróż.

2.6 SGD

W rozdziale dotyczącym gradientu prostego ukryliśmy jedną ważną informację. Chodzi nam tu o sposób obliczania gradientu. Aby obliczyć gradient, potrzebujemy pewnego przykładu, który da nam przestrzeń, nad którą będziemy mogli optymalizować nasz punkt. Jednak często spotykamy się z sytuacją, gdy nie chcemy dopasować naszej wartości do jednego przykładu, ale jest tych przykładów więcej. Pomocny w tym procesie jest zbiór danych.

Ryc. 2.6: mini-batch SGD



Zbiór danych jest kolekcją pewnych danych, które mają ze sobą coś wspólnego. Np. możemy mówić o zbiorze zdjęć ze ślubu lub o zbiorze nazw książek na twojej półce albo o zbiorze cen pewnych produktów.

$$D = [x_1, x_2, x_3, \dots] \tag{2.16}$$

Możemy sobie wyobrazić sytuację, w której chcielibyśmy, powiedzmy znaleźć najlepsze położenie nie tylko dla naszego domu, ale też dla wybudowania całego osiedla mieszkaniowego. Musimy wtedy wziąć pod uwagę preferencje wszystkich naszych klientów, a nie tylko nasze własne. Używając zbioru danych, moglibyśmy przechowywać preferencje dla wszystkich klientów. Aby zrobić jeden krok naszego algorytmu gradientu prostego, musielibyśmy obliczyć gradient, używając wszystkich przykładów ze zbioru danych, więc liczba obliczeń wzrosłaby proporcjonalnie do wielkości zbioru danych. W tym wypadku gradient dla pojedynczego kroku byłby sumą gradientów dla każdego przykładu. Możesz sobie wyobrazić, że może to być dość niepraktyczne, jeśli zbiór składa się z tysięcy, a nawet milionów przykładów. Jak rozwiązać ten problem, zmniejszając liczbę koniecznych obliczeń, a jednocześnie zachowując generalizację? Rozwiązaniem jest tu wykonanie pojedynczego kroku z jednym przykładem, a następnie przejście do innego przykładu i wykonanie kroku już na nim. Tym sposobem w pewnym momencie korzystamy z informacji na temat każdego przykładu, jednocześnie oszczędzając na obliczeniach. Właściwie takie podejście nazywa się **metodą gradientu stochastycznego** (ang. stochastic gradient descent) a krótko SGD. Algorytm ten otrzymuje swoją nazwę od tak zwanej stochastyczności, czyli matematycznej nazwy na procesy losowe. W tym przypadku nazwa związana jest z losowością powstającą na skutek używania pojedynczych przykładów, które mogą być niereprezentatywne dla całego zbioru danych. Widzimy więc, że SGD ma też swoje wady. Jakie są główne różnice pomiędzy używaniem całego zbioru danych a używaniem pojedynczego przykładu? Trenowanie na całym zbiorze danych jest dokładniejsze, ponieważ różnice między pojedynczymi przykładami się niwelują, ale jest również bardziej kosztowne obliczeniowo. Natomiast używanie pojedynczego przykładu przeciwnie: jest mniej kosztowne obliczeniowo, ale za to dużo bardziej niedokładne. Może więc moglibyśmy znaleźć rozwiązanie, tak aby spotkać się w połowie drogi. Trenujmy na kilku przykładach wybranych z naszego zbioru danych. To podejście nazywa się **mini-batch** co z angielskiego oznacza „mały pakiet". Algorytm nazywa się mini-batch SGD. Czasami dla skrótu mówi się na niego SGD, opuszczając pierwszy człon, co może być mylące dla osoby niezaznajomionej z tematem. Takie podejście zniweluje fluktuacje w funkcji wartości, które możemy obserwować w czystym SGD i sprawi, że uczenie będzie łatwiejsze. Ten algorytm jest jednym z najczęściej stosowanych w AI. Nawet pomimo tego, że był on jednym z pierwszych, które zostały wymyślone, to nadal jest często używany i potrafi osiągnąć najlepsze wyniki w trenowaniu np. sieci neuronowych.

Rozdział 3

Sieci neuronowe

Sieci neuronowe są jednym z najgorętszych tematów w świecie sztucznej inteligencji a może także poza nim. Mają one proste i eleganckie właściwości, które zostały zainspirowane myśleniem o podobnych systemach znajdujących się w mózgu. Pomimo tego, że ich genezą są modele mózgu, to teraz wiemy, że różnią się one od ich biologicznych odpowiedników na pewne ogromne sposoby. Na przykład prawdziwe neurony są aktywowane inaczej i mają prawdopodobnie inny mechanizm uczenia się. Ponieważ wiedza o mózgu jest ograniczona, nie możemy wiedzieć, czy tak jest na pewno. Czy te różnice są do pogodzenia, nie jest jeszcze pewnym. Wiemy, natomiast że sieci, które działają na naszych komputerach, mogą osiągać dosyć niesamowite wyniki. Jednak nie było tak zawsze. Jeszcze kilkadziesiąt lat temu wiele osób miało wątpliwości na temat tego, czy sieci neuronowe są w stanie dokonać niektórych rzeczy. To tak jakbyśmy przenieśli się w czasie do wynalazku silnika. Dla wielu osób w tym czasie nie było zapewne jasne, do czego można go użyć, a nawet jeśli zdawali sobie sprawę z niektórych zastosowań, to nie byli sobie w stanie wyobrazić innych. Podobnie na początku badania sztucznej inteligencji nie było wiadomo, co da się osiągnąć, a co jest, niemożliwe stosując te metody. Nie wiedziano nawet które problemy są łatwe a które ciężkie do rozwiązania. Niektórzy myśleli, że rozwiązanie problemów, które są ciężkie dla nas, takich jak różniczkowanie, będzie ciężkie również dla maszyn, a problemów łatwych dla nas, jak rozpoznawanie obiektów, będzie dla nich łatwe. Okazało się inaczej. Problem różniczkowania został rozwiązany dość szybko przez Jamesa Slagle'a, który napisał pierwszy program obliczający całki. Co ciekawe Slage był niewidomy, nie przeszkodziło mu to jednak w napisaniu programu całkującego w języku LISP. LISP był początkowo popularnym językiem programowania w kręgach ludzi zajmujących się sztuczną inteligencją. Był, można nawet powiedzieć, elitarnym językiem programowania. Dzięki swojej prostej i eleganckiej składni, która przypominała operowanie na listach, nowoczesnym rozwiązaniom oraz przede wszystkim traktowaniu każdego wyrażenia jak symbolu stał się on popularny w kręgach AI. LISP był też nietypowy, gdyż niektóre wyrażenia, które można naturalnie zapisać w innych językach, trzeba było napisać w sposób rekursywny, co stanowiło dodatkową trudność. LISP nie był językiem stworzonym dla sieci neuronowych. Jego wolne działanie w porównaniu do innych języków tamtego czasu, takich jak C, było przeszkodą dla algorytmów wymagających szybkości

obliczeniowej. W tamtym czasie wielkich przełomów nie to było najważniejsze. Konceptualne przełomy dokonywały się jeden po drugim, a naukowcy dokonywali wielkich obietnic, ponieważ widzieli ogromny postęp w kilku formalnych dziedzinach. Problemy takie jak rozpoznawanie obiektów nadal pozostawały nierozwiązane. Kiedy okazało się niemożliwym, żeby rozwiązać inteligencję bez wcześniejszego poradzenia sobie z tymi 'łatwiejszymi' problemami przyszła tzw. zima AI (ang. AI winter), kiedy to fundusze na badania zostały ograniczone. Wszyscy byli zawiedzeni tym, co zostało osiągnięte, ponieważ spodziewano się jeszcze szybszych postępów, jednak ukryte problemy dawały znać. Dotknięta została również dziedzina badań nad sieciami neuronowymi, które w tamtym czasie nie dawały tak niezwykłych rezultatów. W konsekwencji tylko kilku najwytrwalszych naukowców kontynuowało badania nad sieciami neuronowymi. Byli to naukowcy z Kanady i jeden Francuz. W 2019 za badania prowadzone w czasie kiedy nikt nie wierzył w sieci neuronowe Hinton, Bengio i LeCun zostali nagrodzeni nagrodą Turinga. Istniał też inny realny problem poza wygórowanymi oczekiwaniami. Komputery lat 70 były dużo wolniejsze niż obecne. A jeśli jest jedna rzecz, która jest prawdziwa w stosunku do sieci neuronowych to to, że są ogromnie głodne zasobów. Było po prostu niemożliwym, żeby osiągnąć niektóre z niezwykłych rezultatów, które potrafimy dokonać obecnie na tamtych komputerach. Dzisiaj sieci neuronowe są używane we wszelakich rozwiązaniach, zaczynając od rozpoznawania liter, znaków, twarzy, przeprowadzania diagnozy medycznej, grania w gry komputerowe, szachy i Go, po wykonywanie za nas nużących czynności jak filtrowanie spamu czy wybieranie kolejnego filmu do odtworzenia albo piosenki do posłuchania.

3.1 Perceptron

Pomysł stojący za **perceptronem** jest prosty. Weźmy kilka danych wejściowych, połączmy je w pewien sposób i zwróćmy wartość numeryczną. Jeśli ta wartość jest większa niż zero, to klasyfikujemy dane wejściowe jako należące do klasy 0, w przeciwnym wypadku klasyfikujemy je jako klasę 1. Na ten pomysł wpadł Frank Rosenblatt w 1958 roku. Perceptron początkowo miał być nie programem a specjalną elektroniczno-elektro-mechaniczną maszyną zafundowaną przez amerykańskie wojsko. Rosenblatt 'sprzedał' wojskowym swój pomysł jako maszynę do rozpoznawania znaków na zdjęciach, choć oni wiązali z nim prawdopodobnie dużo większe nadzieje. Pomyślmy co moglibyśmy chcieć klasyfikować. Możemy np. chcieć sprawdzić, czy na zdjęciu jest kot, czy go tam nie ma albo sprawdzić, czy podana litera to 'A', czy nie. Jednak wojskowi pokładali zapewne największe nadzieje w rozpoznawaniu sprzętu wojskowego na zdjęciach. Jak miałoby to działać w perceptronie? Musimy mu najpierw przekazać nasze dane jako wejściowe, np. jeśli chcemy klasyfikować zdjęcia, powinniśmy mu dostarczyć listę wszystkich pikseli. Teraz musimy pomyśleć, jak chcemy połączyć wszystkie informacje, które otrzymuje perceptron. Na początku zmieniamy piksele w związane z nimi

wartości numeryczne, a następnie mnożymy je przez jakąś pozytywną lub negatywną wagę. Kolejno musimy tylko dodać wszystkie wartości, które otrzymaliśmy i sprawdzić, czy ich suma jest większa, czy mniejsza od 0.

$$\begin{aligned} f(x) = 1, \text{jeśli} \;\; w * x > 0 \\ \text{i} \;\; f(x) = 0 \text{ w przeciwnym wypadku} \end{aligned} \tag{3.1}$$

W równaniu (3.1) $\boldsymbol{w}$ są wagami, przez które mnożymy dane wejściowe $\boldsymbol{x}$, następnie, jeśli suma jest większa niż 0, zwracamy jeden, inaczej zwracamy zero.

Działanie perceptronu możemy zwizualizować jako oddzielanie płaszczyzny prostą linią. Z jednej strony takiej linii będą przykłady pozytywne, czyli te oznaczone jako 1, a z drugiej negatywne, czyli oznaczone jako 0. Ta linia będzie koniecznie przechodzić przez punkt (0, 0). Możesz to sprawdzić, wybierając jakieś wagi i wstawiając do równania $\boldsymbol{x}$ równe 0. Jeśli nie lubimy tej właściwości, a nie jest ona najbardziej pożądana ze względu na niemożliwość rozdzielenia pewnych zbiorów danych, to możemy dodać tzw. **współczynnik korygujący** (ang. bias) do każdej danej wejściowej. Więc:

$$\begin{aligned} f(x) = 1, \text{jeśli} \;\; w * x + b > 0 \\ f(x) = 0 \;\; \text{w przeciwnym wypadku} \end{aligned} \tag{3.2}$$

Jedyna zmiana następuje w pierwszej części równania (3.2), gdzie dodajemy wartości $\boldsymbol{b}$ do każdej danej wejściowej. I pamiętamy tu, że dodawanie następuje po mnożeniu.

Co pozostało nam do zrobienia? Potrzebujemy jakoś określić wartość naszych wag, ale jest bardzo dużo możliwości, na które moglibyśmy to zrobić. Wagi mogą w teorii przyjmować wartość równą dowolnej liczbie rzeczywistej. Jednak aby poprawnie klasyfikowały nasze dane, konieczne jest, aby były odpowiednio dobrane. Tu z pomocą przychodzi rozdział 2.X. Możemy po prostu użyć algorytmu SGD na przykładach zdjęć z kotami. Żeby mogło to jednak zadziałać, potrzebna jest nam funkcja wartości, która będzie określać, jak dobrze zaklasyfikowaliśmy dany przykład. Jak dokładnie będzie wyglądać nasza funkcja wartości w tym przypadku? Funkcją wartości będzie dystans pomiędzy przykładem a **granicą decyzyjną**, tutaj naszą prostą $\boldsymbol{f(x) = 0}$. Dystans ten pomnożymy przez 1, jeśli zaklasyfikowaliśmy przykład poprawnie i -1, jeśli zrobiliśmy to błędnie. Zauważmy, że wartość funkcji wartości zależy od odległości przykładu od prostej. Sprawia to, że perceptron próbuje znaleźć najlepszą taką prostą, a także pomaga w uczeniu, poprzez pokazywanie jak daleko nasz przykład znajduje się od bycia źle zaklasyfikowanym. Pozostaje nam dodać ostatnią część. Moglibyśmy, co prawda, zrobić to, co opisaliśmy, ale byłoby koniecznym używać wszystkich przykładów do obliczania jednego kroku. Dzieje się tak, ponieważ nie mamy sposobu, aby zmienić

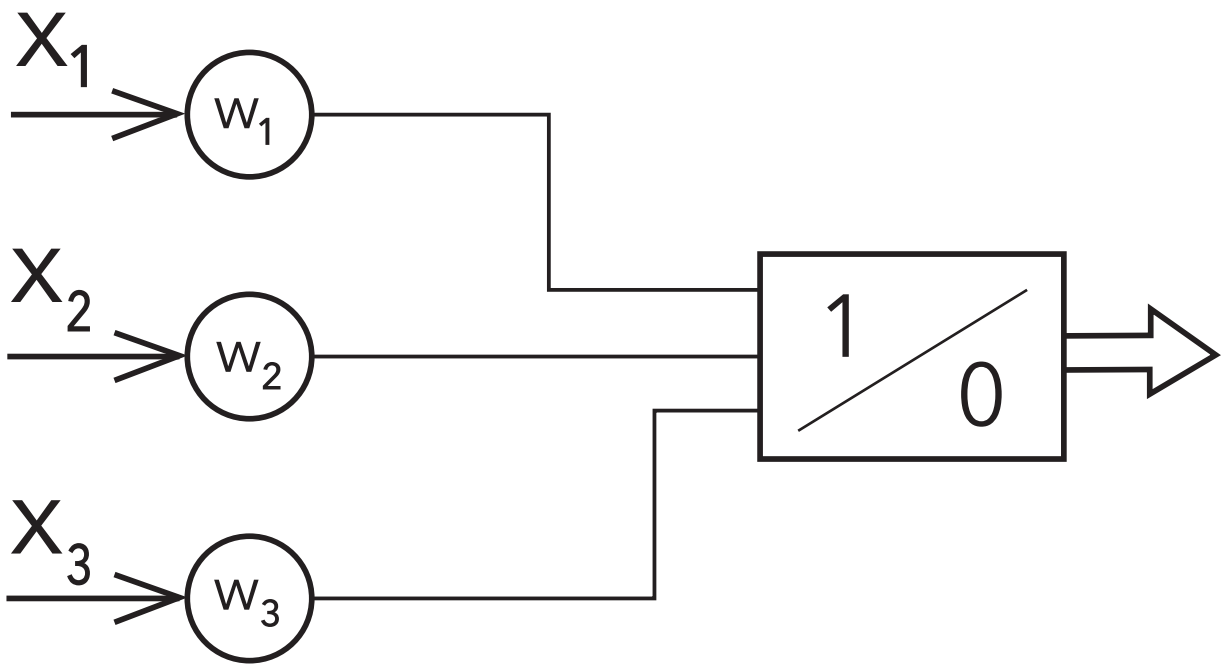
wynik klasyfikacji za pomocą pojedynczego przykładu. Może nam się błędnie wydawać, że SGD rozwiązuje ten problem. Możemy przecież trenować po kolei na kolejnych przykładach, jednak musimy powiedzeć, że SGD nie wykorzystuje informacji, o tym, jak mocno dany piksel się 'świeci' do poprawy klasyfikacji. A zauważmy, że obecność danego piksela o określonym kolorze może świadczyć o obecności np. kota. SGD z funkcją wartości, którą opisaliśmy, spróbuje tylko znaleźć odpowiednie wagi, które prowadzą do najlepszej klasyfikacji, nie weźmie natomiast pod uwagę jakie elementy wejściowe przyczyniły się do osiągnięcia rezultatu. SGD zmienia wagi na korzystniejsze, patrząc się na przykłady tylko aby określić funkcję wartości, poza tym skupia się na zmienialnych parametrach czyli wagach. Chcemy aby nasze rozwiązanie nie zmieniało, w sposób ślepy wag, tylko brało pod uwagę to, w jaki sposób dane wejściowe mogą być skorelowane z klasyfikacją. Na pomoc przychodzi **reguła aktualizacji** perceptronu (ang. update rule). Jest ona jedną z najciekawszych jego części. Mówi nam ona, że:

$$w_i(t+1) = w_i(t) + (y - (w * x + b)) * x \tag{3.3}$$

Gdzie $\boldsymbol{w_i(t)}$ są wagami w kroku $\boldsymbol{t}$, $\boldsymbol{y - (w * x + b)}$ jest różnicą między oczekiwanym rezultatem $\boldsymbol{y}$ a otrzymanym $\boldsymbol{w * x + b}$ a $\boldsymbol{x}$ jest wartością wejścia.

Wszystkie elementy mają pewien sens, w świetle tego, o czym mówiliśmy. Zastosowaliśmy regułę rekurencyjną do aktualizacji wag. Do tego zmieniamy je o różnicę między oczekiwanym wyjściem a otrzymanym, używając pojęcia granicy decyzyjnej. Jedyne czego nie wiemy, to, skąd pojawiło się $\boldsymbol{x}$ na końcu równania. Możemy pomyśleć o sytuacjach, w których chcielibyśmy zwiększyć zmianę, którą aplikujemy do naszych wag. Co się dzieje, gdy zwiększamy ją gdy $\boldsymbol{x}$ jest duże i zmniejszamy gdy $\boldsymbol{x}$ jest małe? Zmiana będzie większa przy większych wartościach tej zmiennej, ale całkowita zmiana zależy też od różnicy pomiędzy spodziewanym a otrzymanym wynikiem, czyli od błędu przewidywania. A więc zmiana wag będzie wzmacniana, gdy jednocześnie błąd i wejście będą duże. Jeśli taka sytuacja ma miejsce, to znaczy, że $\boldsymbol{x}$ jest dobrym predyktorem tej różnicy, a więc waga odpowiadająca temu $\boldsymbol{x}$ powinna być zmieniona tak, aby była bardziej receptywna na to wejście. To właśnie osiąga reguła aktualizacyjna perceptronu. Jest to jeden z najważniejszych pomysłów dotyczących sieci neuronowych. Używając tej reguły, szukamy danych wejściowych, które dobrze przewidują błąd, a więc pomagają w otrzymaniu poprawnego wyniku.

Рис. 3.1: Perceptron



Czy wiedząc wszystko, co zostało zawarte w tym rozdziale, możemy iść i próbować klasyfikować koty? Prawdopodobnie nie. Mimo iż perceptron był zaprojektowany, by klasyfikować obrazy, jest on dość zły w tym zadaniu. Jedyne co robi perceptron, to tworzy oddzielającą przykłady płaszczyznę w $\boldsymbol{n}$ wymiarowej przestrzeni. Taka płaszczyzna oddzielająca jest zbyt mało skomplikowana, aby przedstawić większość różnic między przykładami np. kotów. Perceptron może być bardziej predysponowany, do powiedzmy, klasyfikacji domów na te zawierające basen i te go nie posiadające, używając do tego celu informacji na temat ceny, metrażu i dystansu do centrum miasta. Jeśli chodzi natomiast o bardziej złożone zastosowania, perceptron jest przestarzały, mając na uwadze obecne standardy. Może on mieć dosyć duże problemy z pewnymi specjalnie wybranymi, ale bardzo prostymi funkcjami takimi jak XOR (XOR jest funkcją która przyjmuje dwa argumenty, oba z nich są 0 albo 1, i zwraca 1, jeśli wejścia są różne i 0 w przeciwnym przypadku). W roku 1969 Marvin Minsky i Seymour Papert wydali słynną książkę o nazwie „Perceptrons", która matematycznie udowadniała, że perceptron nie jest w stanie działać jak bramka XOR, co spowodowało w tamtym czasie spadek zainteresowania rozwiązaniami takimi jak perceptron, ponieważ wydawało się, że udowodniono, że sieci typu perceptron mają pewne nierozwiązywalne ograniczenia z nimi związane.

3.2 Neurony

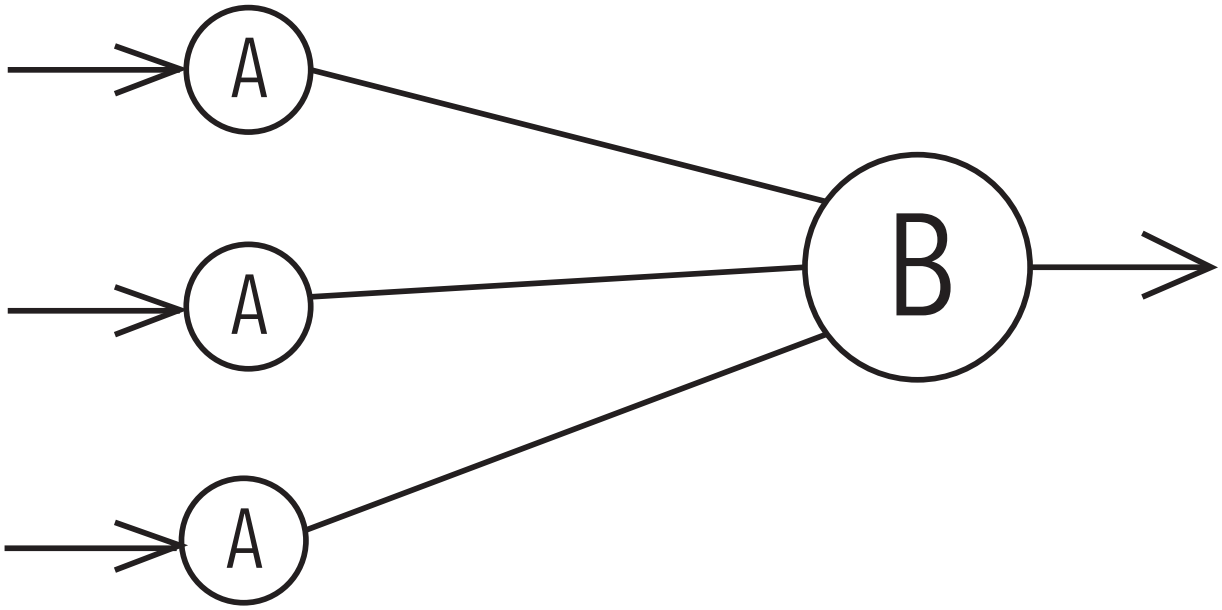
O neuronach wiemy, wydaje się nam, intuicyjnie dużo, przecież towarzyszą nam każdego dnia. Jednak niestety żaden z nich nie raczył się nam przedstawić i objaśnić sposobu swojego działania. Pomogłoby to nam zapewne w podejmowaniu lepszych decyzji, jeśli wiedzielibyśmy, z jakich części składa się nasz mózg. Jednak przy braku takich kurtuazji pozostaje się nam zdać na nasze własne badania i dociekania na ten temat. W poszukiwaniu wyjaśnienia wewnętrznego działania mózgu ludzie stworzyli modele tego, co jest w środku. Jeden z najprostszy takich modeli był używany w perceptronie, który opisaliśmy poprzednio. Nie będziemy tu opisywać takich modeli prawdziwych neuronów ze względu na ortogonalność takiego przedsięwzięcia oraz brak wiedzy autora na ten temat. Co jednak zrobimy, to opiszemy modele używane w sztucznych sieciach neuronowych. Należy zaznaczyć, że te modele nie są identyczne i, według naszej wiedzy, różnią się od siebie w znaczący sposób. Przejdźmy więc do opisu wytworu naszej wyobraźni. Każdy sztuczny neuron posiada pewną liczbę $\boldsymbol{n}$ wejść, które zazwyczaj są liczbami rzeczywistymi. Celem takiego neuronu jest dokonanie pewnej transformacji tych danych wejściowych. Możemy niejako myśleć o działaniu pojedynczego neuronu jak o wyniku działania perceptronu, który klasyfikuje dane, które otrzymuje na pewne klasy. To był najpopularniejszy pogląd, na to, jak powinien działać neuron jeszcze kilkadziesiąt lat temu. Zazwyczaj klas, na które neuron miał

klasyfikować wejście, było dwie. Jedna dla stanu aktywnego a druga dla stanu nieaktywnego. W ten sposób miała być oddana właściwość prawdziwych neuronów, które albo są, albo nie są aktywne w zależności od wejścia. W następnym podrozdziale zobaczymy jak połączenie neuronu z funkcją aktywacji, może dać podobny wynik. Jednak podstawową różnicą między perceptronem a neuronami jest to, że te drugie, jak możesz się domyślać, łączymy w większe sieci, składające się z wielu neuronów. Te większe sieci są podstawą potęgi neuronów. Tak jak to zostało udowodnione w komputerach, u mrówek czy u ludzi, nie siła pojedynczej jednostki decyduje o sile systemu, lecz raczej zbiorowe, skoordynowane działanie. Sieci neuronów wykorzystują efekt sieci i skali, aby się wzajemnie wzmacniać. Pomimo że działanie jednego neuronu jest proste i łatwe do opisania to działanie ich grup daje niespodziewane wyniki i jest ciężkie do ujęcia matematycznymi wyrażeniami w sposób przejrzysty dla człowieka. O tym też powiemy w następujących podrozdziałach. Wracając do działania neuronu, zazwyczaj każde wejście $\boldsymbol{x}$ zostaje na początku pomnożone przez wagę $\boldsymbol{w}$, następnie dodajemy współczynnik korygujący $\boldsymbol{b}$, zupełnie jak w perceptronie, i sumujemy wszystkie rezultaty.

$$u = \Sigma(x * w + b) \tag{3.4}$$

Sumując wszystkie wartości $\boldsymbol{x * w + b}$ otrzymujemy pewien wynik $\boldsymbol{u}$. Moglibyśmy go uznać za wynik działania neuronu, jednak to znaczyłoby, że wynikiem mogłaby być dowolna liczba, co niekoniecznie jest wskazane. Pomyślmy, że taka liczba mogłaby być dowolnie mała lub duża, co sprawiałoby nie tylko problemy z przetworzeniem jej przez komputer, ale także mogłaby niszczyć wyniki działania wielu innych neuronów, jeśli zostałaby użyta wraz z ich wynikami do dalszych obliczeń. Z tego powodu chcemy, aby nasza wartość $\boldsymbol{u}$ została przetworzona przez tak zwaną **funkcję aktywacji**. Jest ona zazwyczaj nieliniowa, ponieważ nie chcemy, aby nasz neuron zachowywał się jak perceptron, to znaczy zawsze dzielił przestrzeń prostą linią. Jeśli użyjemy funkcji nieliniowej jako funkcji aktywacji, to podział przestrzeni będzie mógł być dużo bardziej złożony. Ta nieliniowość wprowadzona przez funkcję aktywacji sprawi, że podział będzie nieliniowy, a więc kilka połączonych ze sobą neuronów będzie mogło stworzyć dużo bardziej skomplikowany podział przestrzeni, niż taki, który moglibyśmy otrzymać bez funkcji aktywacji.

Рис. 3.2: Neuron



3.3 Funkcje aktywacji

Funkcja aktywacji $\boldsymbol{\phi}$ bierze jako wejście $\boldsymbol{u}$ i zwraca wyjście neuronu.

$$y = \phi(u) \tag{3.5}$$

Wszystkie równania na neuron przedstawiają się następująco:

$$u = \Sigma(x * w + b) \tag{3.6}$$

$$y = \phi(u) \tag{3.7}$$

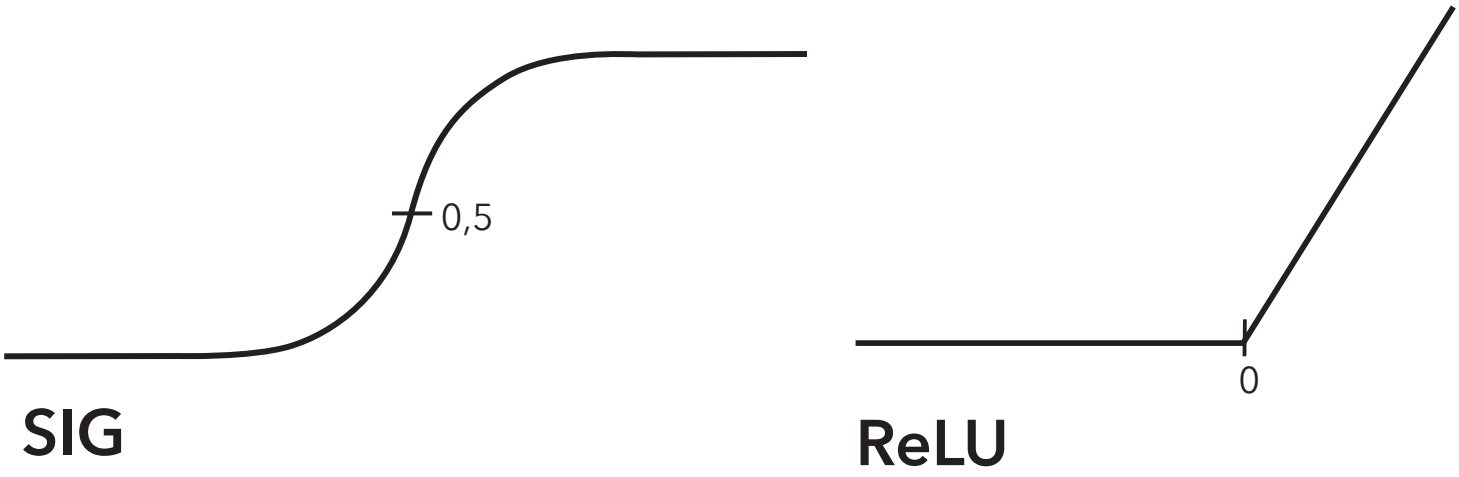
Pozostaje nam tylko dodać jakiej funkcji aktywacji należy używać. Funkcją, która była najbardziej popularna na początku wiedzy o sieciach neuronowych była **funkcja sigmoidalna** zdefiniowana jako:

$$S(x) = 1/(1 - e^{-x}) \tag{3.8}$$

Ta funkcja ma kształt litery 'S'. Zmienia się ona bardzo mało na końcach, a gradient jest największy blisko zera. Zwraca ona wartości pomiędzy 0 a 1, z tym że większość wartości leży blisko 0 lub 1 ze względu na kształt funkcji. Było to początkowo interesującą właściwością ze względu na podobieństwo do neuronów w mózgu, które albo są w stanie aktywacji, albo nie dają żadnego sygnału. Dodatkowym plusem jest nieliniowość tej funkcji. Pochodna funkcji sigmoidalnej jest prosta i jest to niewątpliwie silna strona ponad niektórymi konkurentami. Okazuje się jednak, że jest pewien duży problem z tą funkcją. Staje się ona w łatwy sposób nasycona, co znaczy, że jeśli $\boldsymbol{x}$ zmienia się znacząco to $\boldsymbol{y}$ zmienia się bardzo mało. Nasycenie w funkcji sigmoidalnej ma miejsce przy jej końcach. Kiedy takie nasycone funkcje zostają przez siebie pomnożone z wynikiem znajdującym się blisko zera, sprawia to, że wartość numeryczna staje się bardzo mała. Ponieważ komputery nie radzą sobie z arbitralną dokładnością, prowadzi to do niedokładności w obliczeniach gradientu podczas trenowania sieci neuronowej. Problemy z tego powodu nazywane są **znikającym gradientem** (ang. vanishing gradient). Może też wystąpić przeciwieństwo znikającego gradientu, kiedy duże wyniki funkcji są mnożone przez siebie, to może to prowadzić do eksplozji gradientu. Taka sytuacja jednak nie będzie mieć miejsca przy sigmoidalnej funkcji aktywacji, ponieważ jej wartość jest ograniczona do 1, natomiast przy innych funkcjach nieograniczonych z góry, jest to bardzo realne zagrożenie. Inną często używaną funkcją jest ReLU (ang. rectifier linear unit), ma ona skomplikowaną nazwę i bardzo prostą definicję:

$$R(x) = max(0, x) \tag{3.9}$$

Ryc. 3.3: Funkcje aktywacji



Jej wartość wynosi 0 dla negatywnych wartości $\boldsymbol{x}$ i jest równa dokładnie $\boldsymbol{x}$ dla wartości pozytywnych. W większości sytuacji powinno się używać ReLU zamiast funkcji sigmoidalnej, ponieważ daje ona lepsze empiryczne rezultaty. Ta funkcja ma dodatkową właściwość, będąc bardzo szybką do policzenia, co oszczędza cykle procesora. Możesz sobie myśleć: Poczekaj chwilę, czy ta funkcja nie jest przypadkiem liniowa? Odpowiedź brzmi: nie, nie jest liniowa ze względu na to, że jest ona liniowa tylko w pewnych jej częściach, które są połączone w punkcie (0, 0). Okazuje się, że to wystarcza, aby zapewnić bogate zachowanie dla całej sieci. Istnieją pewne wariacje na temat funkcji ReLU. Krótko opiszemy dwie z nich.

Nieszczelne ReLU (ang. leaky ReLU) jest jednostką ReLU, która jest 'nieszczelna', to znaczy, że jeśli $\boldsymbol{x}$ jest mniejsze od zera, to nie zwraca ona 0 tylko zamiast tego $\boldsymbol{x}$ * 0,01, lub inną małą wartość zależną od $\boldsymbol{x}$.

Parametryczne ReLU (ang. parametric ReLU) jest taki sam jak nieszczelne ReLU z tą małą różnicą, że nie mnożymy $\boldsymbol{x}$ przez 0,01 tylko przez $\boldsymbol{a}$ które jest znalezione w procesie uczenia. To kończy nasz krótki przegląd funkcji aktywacji.

Funkcja sigmoidalna i ReLU stanowią dwie najważniejsze opcje, które mamy do wyboru tworząc model sieci neuronowej. Poza tym istnieją oczywiście inne możliwości, takie jak, chociażby tanges hiperboliczny. Faktem jest natomiast, że zazwyczaj dzielą kluczowe właściwości, silnie oraz słabe strony z jedną z dwóch opisanych tu funkcji. Z tego właśnie powodu ograniczyliśmy listę funkcji aktywacji do dwóch najważniejszych przykładów. Dodajmy, że perceptron posiadał swoją własną inną od tych tu opisanych funkcję aktywacji. Możesz teraz powrócić do rozdziału o perceptronie i zobaczyć czy ją rozpoznasz.

3.4 Wszystko razem

Teraz pokryliśmy wszystkie części budulcowe potrzebne do stworzenia sieci neuronowej. Możesz się siebie teraz pytać: Zaczekaj, czy nie mówiliśmy tylko o neuronach? Tak rozmawialiśmy tylko o neuronach, ale piękną rzeczą dotyczącą sieci neuronowych jest to, że potrzebują jako budulca tylko neuronów. Nie oznacza to, jednak że nie pozostało nam nic do zrobienia. Musimy się zastanowić nad jedną kluczową kwestią. Jak połączymy pewną liczbę neuronów w całość? Jest naprawdę wiele możliwości, na które moglibyśmy tego dokonać, bo każde wyjście neuronu możemy w teorii połączyć z wejściem dowolnego innego neuronu. Moglibyśmy nawet połączyć wyjście naszego neuronu z jego wejściem, tworząc swego rodzaju pętlę zwrotną. Mimo tej dowolności istnieją pewne rodzaje neuronów, które się zwyczajowo wyróżnia. Są

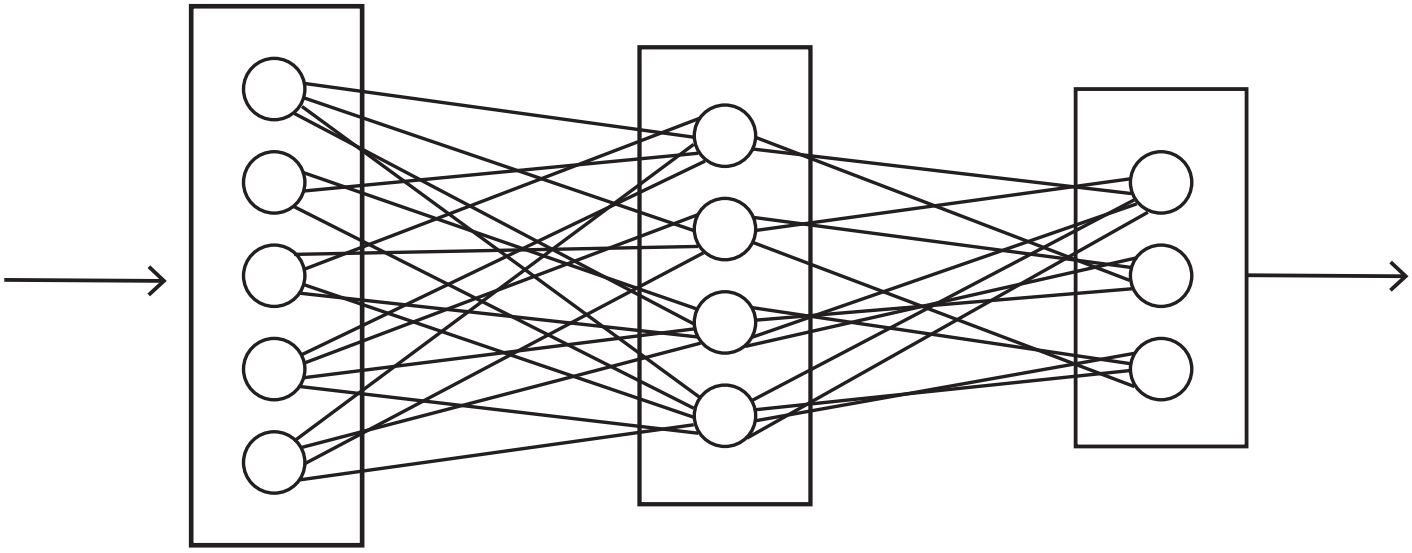
to neurony, które mają trochę inne zastosowania. Nazywane są one **neuronami wejścia** i **wyjścia** (ang. input and output neurons). Widzimy, więc że te same części składowe w sieci neuronowej mogą pełnić różne funkcje. Neurony wejścia różnią się od innych tylko tym, że nie biorą informacji od innych neuronów, ale bezpośrednio z danych wejściowych. Neurony wyjścia za to, są często różnego, specjalnego typu, np. pozostawione bez funkcji aktywacji lub ich wynik jest tak zmieniony, aby zwracały dystrybucję prawdopodobieństwa. Pozwala to na swego rodzaju sformatowanie danych wyjściowych, aby nie były one ograniczone przez funkcję aktywacji. Pomyślmy teraz jak w sensowny sposób połączyć wiele neuronów w całość. Jeśli łączylibyśmy je losowo, to część połączeń byłaby odległych, to znaczy znacząco zmniejszałaby odległość na grafie pomiędzy danymi neuronami. Sprawiłoby to, że w sieci nie istniałabo pojęcie lokalności, ponieważ każdy neuron byłby połączony z odległymi, jak i bliskimi neuronami. Zazwyczaj jednak neurony układamy w tak zwane **warstwy** (ang. layers). To znaczy, że są zapakowane jeden obok drugiego w pewnej kolejności.

$$warstwa = [n_1, n_2, n_3, \dots] \tag{3.10}$$

W jednej sieci neuronowej możemy mieć wiele takich warstw. W tym momencie pojawia się oczywiste pytanie: Jak wiele ma być takich warstw i w jaki sposób mają być połączone? Niestety na pierwsze pytanie musimy odpowiedzieć tak jak poprzednio w takich sytuacjach. To zależy od sytuacji. Nie ma jednej najlepszej liczby warstw ani liczby neuronów w warstwie. To kolejna rzecz, którą możemy zmieniać, tworząc nasz model. Na pytanie na temat połączenia należałoby właściwie też odpowiedzieć, że to zależy od specyfiki architektury, np. istnieją specjalne rozwiązania przeznaczone do rozpoznawania obrazu. Nazywają się one CNN (ang. convolutional neural network) i wykorzystują konwolucje. O konwolucji możemy myśleć jak o pewnym okienku z określonymi wagami dla różnych pozycji w okienku. Takie okienko jest przesuwane przez obraz tak, że w różnych momentach zapisywane są wyniki z różnych części obrazu przetworzonej przez konwolucję. Te wyniki tworzą jakby nowy obraz, na których znowu używane są konwolucje. Wagi w konwolucjach podlegają normalnemu uczeniu podobnie jak w normalnych sieciach neuronowych. Takie konwolucje mają w teorii znajdywać pewne powtarzające się elementy na zdjęciu, takie jak krawędzie, rogi a w wyższych warstwach może np. rozpoznają części ciała takie jak oko, nos itd. Innym typem sieci neuronowej jest tak zwana **losowo połączona** sieć (ang. randomly wired network). Znaczy to, że neurony w pierwszej warstwie łączą się z losowymi neuronami w warstwie drugiej. Kolejne warstwy połączone są w ten sam sposób. Mimo iż moglibyśmy na tym zaprzestać, to nie wiedzielibyśmy czy neurony są połączone w najlepszy sposób. Najlepsze połączenie neuronów jest bardzo skomplikowanym problemem. Skąd w końcu mamy wiedzieć która informacja, z którego neuronu może się przydać dalej? Bardzo prostym rozwiązaniem jest wcale nie zastanawiać się nad tym problemem i neurony w kolejnej warstwie połączyć ze wszystkimi neuronami w warstwie poprzedzającej. W

ten sposób każdy neuron w warstwie otrzymuje wszystkie informacje obecne w warstwie poprzedniej. Jest to nazywane **siecią w pełni połączoną** (ang. fully connected network). Jeszcze innym często wykorzystywanym typem sieci jest RNN (ang. recurrent neural network). Wspomnieliśmy już, że możemy w teorii podłączyć neurony do samych siebie. Tworzy to najprostszą sieć rekurencyjną. Sieci rekurencyjne są przydatne, gdy chcemy, żeby sieć była w stanie przechowywać jakieś dane. Zwykła sieć zwana jest feedforward network, co możemy luźno przetłumaczyć na „sieć przesyłu dalej", jak sama nazwa wskazuje, przesyła tylko informacje do przodu, nie przechowując w pamięci żadnych informacji. Moglibyśmy zamknąć i ponownie otworzyć taką sieć pomiędzy klasyfikowaniem przykładów i otrzymać taką samą odpowiedź. Inaczej jest z RNN, zachowuje ona informacje pomiędzy przykładami, tak że informacja pokazana kiedyś może mieć wpływ na obecny wynik jej działania. Sieć RNN wymaga innego algorytmu propagacji wstecznej (o algorytmie propagacji wstecznej powiemy w następnym rozdziale), zwanego propagacją w czasie. Jeśli kiedykolwiek słyszałeś o sieci nueronowej, to mogło ci się wydawać, że musi to być coś bardzo skomplikowanego. Przecież proste rzeczy nie mogą być tak gorącym tematem debaty. Jednak jak sam widzisz budowa sieci neuronowej, jest bardzo prosta do matematycznego zdefiniowania i nie kryje w sobie wielkich tajemnic. To, co jest naprawdę skomplikowanym, to dopasowanie sieci do problemu, co wymaga szerokiej wiedzy i pewnego rodzaju myślenia przez analogię. Widzieliśmy, że do dobrania jest wiele parametrów, które teoretycznie mogą wpływać na wynik działania, choć zazwyczaj nie będą stanowiły żadnej różnicy. Pozostała nam teraz ostatnia, może najbardziej skomplikowana, część wiedzy, a mianowicie omówienie trenowania sieci neuronowych.

Ryc. 3.4: Sieć neuronowa



3.5 Propagacja wsteczna

Przed dyskusją na temat propagacji wstecznej powinniśmy omówić krótko, czym jest **propagacja** przez sieć neuronową (ang. forward pass). Przy każdej propagacji zaczynamy od podania sieci informacji wejściowych. Następnie informacje są przetwarzane przez pierwszą warstwę i informacje z niej są przesyłane do następnej warstwy do ponownego przetworzenia. Ten proces występuje dla wszystkich warstw, a dane przesuwają się od warstwy wejścia do warstwy wyjścia. Gdy ostatnia, wyjściowa warstwa zostaje osiągnięta, to wynik jest zwracany, a faza propagacji zostaje zakończona. Teraz stworzyliśmy pierwszą sieć neuronową, ale jest jeden problem, który występował również w perceptronie. Żeby nasza sieć robiła coś przydatnego, musimy znaleźć odpowiednie wagi, które dają poprawną odpowiedź. Przypomnijmy o jakich wagach mowa. Każdy neuron przetwarza informacje poprzez pomnożenie wejścia przez $\boldsymbol{w}$ wagi i dodanie $\boldsymbol{b}$ współczynnika korygującego a następne przetworzenie ich przez funkcję aktywacji. Każda waga oraz współczynniki korygujące dla każdego neuronu mogą być ustawione. O ustawianiu tych wag możemy myśleć nie jak o wybieraniu ich dla poszczególnych neuronów a zamiast tego, jak o ustawieniu wag dla całego modelu. Moglibyśmy po prostu stworzyć listę wag wszystkich neuronów i następnie spróbować ustawić je wszystkie naraz za pomocą algorytmu SGD. Przestrzeń, nad którą byśmy optymalizowali, miałaby ilość wymiarów równą ilości sumy wag wszystkich neuronów. Funkcją wartości mogłaby być odległość pomiędzy klasą, którą wybrała sieć a poprawną klasą. Jeśli sieć jest odpowiednio mała to, to podejście może odnieść sukces. Jeśli jednak naszą sieć tworzą tysiące neuronów, wydaje się dość mało prawdopodobnym, żeby wyszukiwanie w takiej wielowymiarowej przestrzeni mogło znaleźć odpowiednie wagi wszystkich neuronów. Pomyślmy o tym jak o strojeniu instrumentu. Łatwo byłoby nastroić instrument, oczywiście z odpowiednimi umiejętnościami, jeśli do zmiany jest tylko jedno 'pokrętło', jeśli jednak tych pokręteł jest kilkadziesiąt i na dodatek ustawiamy je wszystkie naraz, zamiast pojedynczo, to zadanie wydaje się prawie niemożliwe do wykonania. Przypomnijmy sobie, jak rozwiązany był podobny problem w perceptronie. Równanie (3.3) prezentujące regułę aktualizacji w perceptronie wygląda następująco:

$$w_i(t+1) = w_i(t) + (y - j(x)) * x \tag{3.11}$$

Gdzie $\boldsymbol{w_i}$ były wagami, $\boldsymbol{x}$ było wejściem, a $\boldsymbol{j(x)}$ zdefiniujemy jako równe wyjściu $\boldsymbol{w * x + b}$ perceptronu.

Możemy sobie wyobrazić użycie tej samej reguły do zmiany wag w sieci neuronowej. Znamy przecież $\boldsymbol{w}$, $\boldsymbol{x}$ oraz $\boldsymbol{j(x)}$ bo to po prostu wyjście neuronu. Skąd jednak weźmiemy spodziewaną wartość $\boldsymbol{y}$ dla każdego neuronu? Na pewno możemy ją dostać, dla neuronów, dla których zarówno spodziewany, jak i otrzymany wynik jest znany,

czyli dla neuronów wyjścia. Problematyczne są jednak neurony znajdujące się w środkowych warstwach. Przesyłają one przecież informacje tylko do kolejnej warstwy, a przecież ta kolejna warstwa nie jest w stanie podać nam spodziewanego wyniku dla poprzedzających neuronów, bo spodziewana wartość jest znana dopiero dla ostatniej warstwy. Widzimy teraz, dlaczego sieci neuronowe z jedną warstwą są prostsze do stworzenia. Właśnie takie rozwiązania były początkowo wykorzystywane mimo swoich problemów, takich jak problem XOR. Byłoby jednak świetnie, gdyby istniała metoda przesyłania wartości oczekiwanej w głąb sieci, do wcześniejszych warstw neuronów. To pozwoliłoby nam na trenowanie wielowarstwowej sieci. I okazuje się, że taka metoda istnieje. Nazywa się **procedurą propagacji wstecznej** (ang. backpropagation). Używa ona zasady łańcuchowej, która jest formułą obliczania pochodnej funkcji złożonej. Zasada łańcuchowa mówi po prostu, że:

$$dx/dy = dx/dz * dz/dy \tag{3.12}$$

Co może być ubrane w słowa w następujący sposób: pochodna (tu: zmiana) $\boldsymbol{x}$ względem $\boldsymbol{y}$ jest równa pochodnej $\boldsymbol{x}$ względem $\boldsymbol{z}$ pomnożonej przez pochodną $\boldsymbol{z}$ względem $\boldsymbol{y}$. Pozwala to na rozbicie jednej pochodnej na dwie inne.

Nasza zasada aktualizacji dla metody gradientu (2.6) jest równa:

$$w_i(t+1) = w_i(t) - n * dE/dw \tag{3.13}$$

Gdzie $\boldsymbol{w_i}$ są naszymi wagami, $\boldsymbol{n}$ jest stałą uczenia, a $\boldsymbol{dE/dw}$ jest pochodną błędu po wielkości wagi.

Aby obliczyć nowe wagi dla naszej sieci koniecznie potrzebujemy znać **dE/dw**, lecz problemem jest to, że zazwyczaj nie możemy obliczyć tej pochodnej bezpośrednio dla neuronów znajdujących się na dowolnej głębokości w sieci. Zamiast tego użyjemy regułę łańcuchową. Możemy dzięki niej obliczyć pochodną funkcji cząstkowej, jeśli tylko znamy pochodną funkcji złożonej oraz równania wszystkich funkcji. Skupmy się na następującym przykładzie sieci neuronowej składającej się z trzech neuronów, dla którego chcemy obliczyć **dE/dw** dla neuronu **f**:

$$y = h(f(w_1) + g(w_1)) \tag{3.14}$$

Załóżmy też, że znamy błąd określony równaniem:

$$e = E(h(z)) \tag{3.15}$$

Teraz aby obliczyć pochodną błędu dla **z** korzystamy z równania (3.12). Zapiszmy pochodną:

$$dE/dw_1 = dE/dh * (dh/df * df/dw_1 + dh/dg * dg/dw_1) \tag{3.16}$$

Co wynika po prostu z reguł różniczkowania. Użyliśmy reguły łańcuchowej oraz zasady obliczania pochodnej z sumy funkcji.

Znając podane poniżej równania możemy obliczyć ich pochodne:

$$\begin{aligned} e &= E(h) \\ y &= h(f+g) \\ f(w_1), g(w_1) \end{aligned} \tag{3.17}$$

To dawałoby nam możliwość podstawienia odpowiednich wyników do równania (3.16), obliczenie **dE/dw** i wstawienie rozwiązania do równania na aktualizację wag neuronów (3.13).

Jedynym problemem jest to że nie potrafimy obliczyć tych pochodnych ogólnie, ponieważ zależą one również od konkretnego wejścia które otrzymają te funkcje. Tak więc podstawiając pochodne funkcji (3.17) do równania (3.16) otrzymujemy równanie zależne od wejścia **x** na pochodną błędu po wagach **dE/dw**:

$$dE(w_1, x)/dw_1 = dE/dh * (dh/df * df(w_1, x)/dw_1 + dh/dg * dg(w_1, x)/dw_1) \tag{3.18}$$

Następnie najłatwiejszym sposobem aby uzyskać wyniki numeryczne dla określonych wag w_1 oraz wejścia **x** funkcji **f** jest obliczanie tych pochodnych razem z funkcjami, których wynik jest i tak konieczny do działania sieci. Dzięki temu kończąc propagację, będziemy znali wyniki wszystkich pochodnych cząstkowych **df, dg** koniecznych do obliczenia **dE/dw**. Jeśli obliczymy wcześniej równanie na **dE/dw** to będzie wystarczyło podstawić odpowiednie wartości, żeby uzyskać poszukiwaną zmianę dla naszych wag. To pozwoli nam zaktualizować wagi funkcji **f**.

Przedstawimy teraz wyprowadzenie takiego równania dla w pełni połączonej sieci przesyłu do przodu (fully-connected feedforward net).

Przypomnijmy sobie teraz równanie określające neuron (3.6), (3.7):

$$o = \phi(\Sigma(x * w + b)) \tag{3.19}$$

Gdzie ϕ jest funkcją aktywacyjną, a $\boldsymbol{z = \Sigma(x * w)}$ cząstkowym wynikiem działania neuronu jak to wcześniej zdefiniowaliśmy.

Żeby obliczyć pochodną błędu względem wag, będziemy musieli użyć reguły łańcuchowej dwukrotnie:

$$dE/dw = dE/do * do/dz * dz/dw \tag{3.20}$$

Gdzie $\boldsymbol{dE/do}$ jest pochodną błedu względem wyniku działania neuronu, $\boldsymbol{do/dz}$ jest pochodną wyniku działania neuronu względem jego wyniku sprzed uzycia funkcji aktywacji, a $\boldsymbol{dz/dw}$ jest pochodną wartości neuronu przed funkcją aktywacji względem wagi $\boldsymbol{w}$.

Jedyne co zrobiliśmy, to przepisaliśmy pochodną błędu, bo łatwiej będzie nam znaleźć pochodne z prawej strony równania. Teraz weźmy pod uwagę $\boldsymbol{do/dz}$. Jest, to, jak napisaliśmy pochodna wyniku działania neuronu względem wyniku sprzed użycia funkcji aktywacji. Biorąc więc pod uwagę równanie (3.7), możemy zapisać pochodną funkcji aktywacji:

$$do/dz = d\phi(z)/dz \tag{3.21}$$

Pochodna $\boldsymbol{do/dz}$ jest równa pochodnej funkcji aktywacji. Przypomnijmy sobie, jakie mieliśmy funkcje aktywacji. Pochodne funkcji sigmoidalnej (3.8) i ReLU (3.9) to odpowiednio:

$$dS(x)/dx = S(x)(1 - S(x)) \tag{3.22}$$

$$\begin{aligned} R = 1, &\text{jeśli } x > 0, \\ R = 0, &\text{jeśli } x < 0, \\ \text{i } R = 1 &\text{ albo } 0, \text{ jeśli } x = 0 \end{aligned} \tag{3.23}$$

Tutaj, ponieważ ReLU jest nieróżniczkowalne w punkcie $\boldsymbol{x} = 0$, musimy wybrać wartość pochodnej dla tego punktu sami.

Znamy $\boldsymbol{do/dz}$, przejdźmy więc do pochodnej $\boldsymbol{dz/dw}$ z równania (3.20). Jest ona równa pochodnej wnętrza funkcji aktywacji (3.19):

$$dz/dw_i = d\Sigma(x * w)/dw_i = x_i \tag{3.24}$$

Gdzie $\boldsymbol{dz/dw_i}$ jest pochodną wnętrza funkcji aktywacji względem wag, $\boldsymbol{x_i}$ jest 'i' wejściem neuronu. Dla neuronu w środowej warstwie $\boldsymbol{x_i}$ jest równe $\boldsymbol{o_j}$ wyjściu 'j' poprzedniej warstwy. Więc w generalnym przypadku: $\boldsymbol{x_i} = \boldsymbol{o_j}$, a tylko dla pierwszej warstwy $\boldsymbol{x_i} =$ **i-te wejście**.

Teraz podstawiając równania (3.20), (3.21), (3.24) do równania (3.13) otrzymujemy:

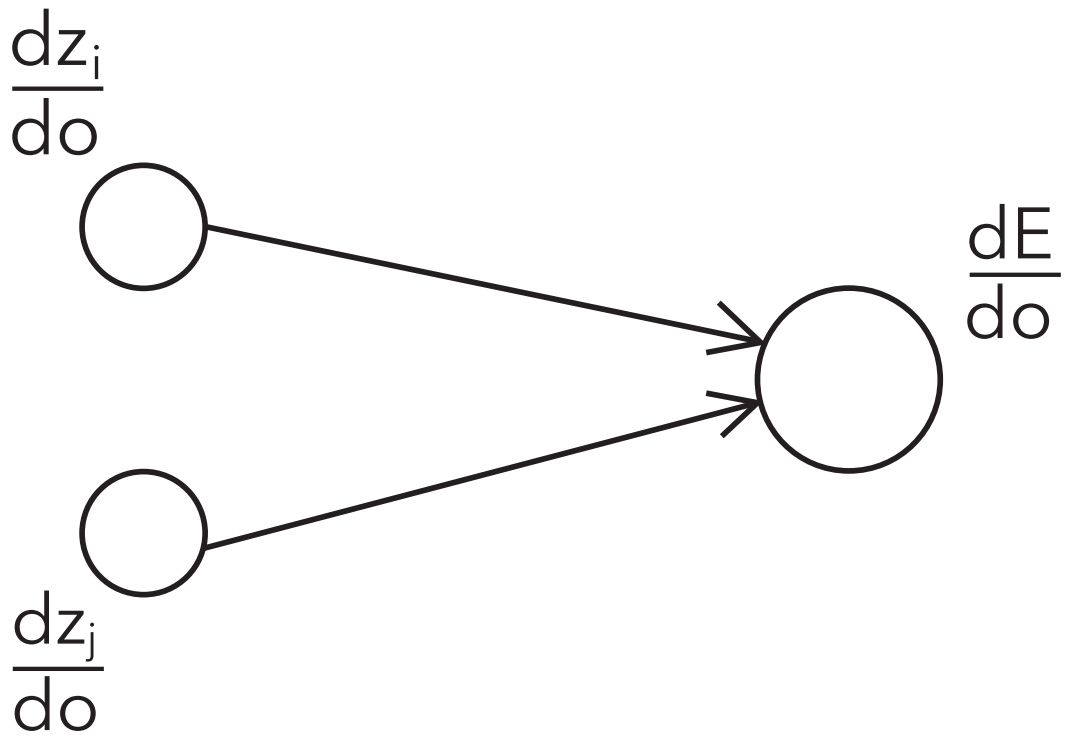
$$w_i(t+1) = w_i(t) - n * dE/do * d\phi(z)/dz * o_j \tag{3.25}$$

To będzie nasze równanie aktualizacji wag. Znamy w nim wszystkie wartości oprócz $\boldsymbol{dE/do}$. Tylko w ostatniej warstwie $\boldsymbol{dE/do}$ jest znane. Dla innych warstw musimy obliczyć tę wartość. Ponieważ nasza sieć jest w pełni połączona, to błąd dla poprzedniej warstwy zależy od wszystkich pochodnych w kolejnej warstwie:

$$dE/do = d(z_i, z_j, z_k, ...)/do \tag{3.26}$$

Gdzie $\boldsymbol{z_N}$ oznacza błąd dla $\boldsymbol{N}$ neuronu w kolejnej warstwie. Przypomnijmy, że: $\boldsymbol{dE/do}$ jest pochodną błedu względem wyniku działania neuronu. Bład ten będzie jednak pochodził od wszystkich neuronów w kolejnej warstwie, ponieważ nasz neuron jest połączony z wszystkimi neuronami w kolejnej warstwie.

Ryc. 3.5: Pochodne w propagacji wstecznej



Teraz biorąc pochodną zupełną równania (3.32), otrzymujemy:

$$dE/do = \Sigma(dE/do_1 * do_1/dz_1 * dz_1/do) \tag{3.27}$$

$$dE/do = \Sigma(dE/do_1 * d\phi(z_1)/dz_1 * w_ij) \tag{3.28}$$

Jedynka na końcu oznacza, że chodzi o następną warstwę. $\boldsymbol{dz_1/do = w_{ij}}$ ponieważ $\boldsymbol{dz_1/do = d\Sigma(w * o)/do = w_{ij}}$. A $\boldsymbol{d\phi(z_1)/dz_1}$ jest pochodną kolejnej warstwy funkcji aktywacji

$\boldsymbol{dE/do_1}$ jest pochodną błędu względem wejścia następnej warstwy funkcji aktywacji, ale jeśli następna warstwa jest warstwą wyjścia, to jak powiedzieliśmy wcześniej $\boldsymbol{dE/do_1}$ jest znane, ponieważ oznacza pochodną błędu względem wyjścia ostatniej wartwy. Teraz, jeśli policzymy $\boldsymbol{dE/do_1}$ dla przedostatniej warstwy i użyjemy go w poprzedniej warstwie, tak jak użyliśmy $\boldsymbol{dE/do_1}$ dla obliczenia $\boldsymbol{dE/do}$ to otrzymamy formułę rekursywną na obliczanie pochodnej błędu względem wyjścia kolejnych poprzednich wyjść neuronów. W algorytmie propagacji wstecznej przesuwamy się od końca sieci do jej początku, obliczając $\boldsymbol{dE/do}$ dzięki formule rekursywnej, używając faktu, że wartość $\boldsymbol{dE/do_1}$ jest znana dla neuronów wyjścia, ponieważ w tym przypadku $\boldsymbol{o = y}$.

$$dE/do_1 = dE(y)/dy \tag{3.29}$$

Dla ostatniej warstwy możemy w łatwy sposób policzyć pochodną błędu względem $\boldsymbol{y}$ używając technik różniczkowania. Jednak jeśli znamy $\boldsymbol{dE/do_1}$ dla warstwy wyjścia to możemy, używając równania (3.28) policzyć $\boldsymbol{dE/do_1}$ dla warstwy przed warstwą wyjścia. Możemy tę czynność powtarzać dla kolejno poprzednich warstw, aż dojdziemy do wejścia. Korzystając z następujących równań, możemy policzyć, w jaki sposób powinniśmy zmienić wagi w sieci:

$$w_i(t+1) = w_i(t) - n * dE/do * d\phi(z)/dz * o_j \tag{3.30}$$

$$dE/do = \Sigma(dE/do_1 * d\phi(z_1)/dz_1 * w_{ij}) \tag{3.31}$$

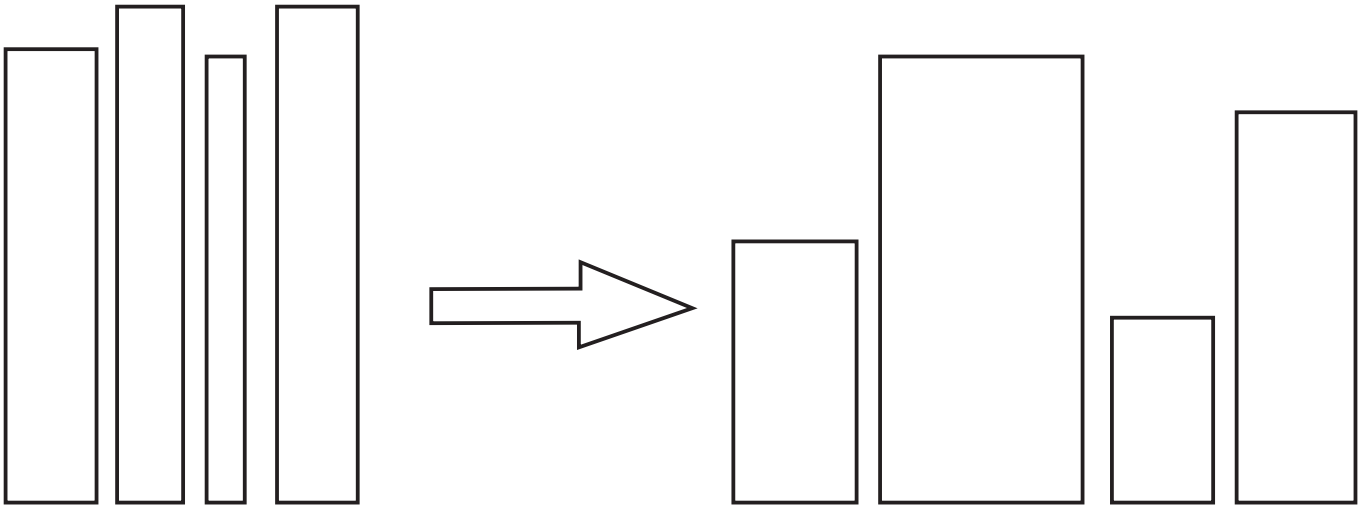
Warto powiedzieć, że równanie (3.30) i (3.31) mogą być wykonywane niezależnie. Np. w bibliotece PyTorch równanie (3.31) dla każdej warstwy jest wykonywane, kiedy wykonywana jest propagacja wsteczna. Równanie (3.30) jest natomiast wykonywane, dopiero kiedy sami podamy taką komendę. Oznacza to, że gradient jest obliczany niezależnie od ustawiania wag.

3.6 Zaawansowane tematy w zagadnienu

W poprzednich podrozdziałach przypatrywaliśmy się bliżej podstawowemu wprowadzeniu do sieci neuronowych. Nauczyliśmy się, jak możemy je zbudować, używać propagując przez nie dane i trenować używając algorytmu propagacji wstecznej. Jeśli nie zrozumiałeś poprzedniego rozdziału, proponujemy tutaj pewną alternatywną, rzadziej używaną metodę trenowania sieci neuronowych. Możemy myśleć o wagach w naszym modelu jak o genomie organizmu. Każda waga koduje w tym modelu określone zachowania. Bardziej sprawny organizm daje większą wartość funkcji wartości. W celu znalezienia lepszych organizmów użyjemy metod programowania genetycznego, krzyżując najlepsze organizmy z nadzieją, że efektem będą jeszcze sprawniejsze jednostki. Następnie przetestujemy je na naszym problemie. Wybierzemy kilka najbardziej sprawnych i znów skrzyżujemy jednostki dające najlepszy wynik. Jest to podejście symulujące ewolucję wykorzystane w celu poprawy naszych wag. Ten sposób trenowania sieci, choć zazwyczaj bardziej wymagający obliczeniowo, jeśli damy mu wystarczająco dużo czasu da nam równowartościowe rezultaty do metody propagacji wstecznej.

Mówiliśmy, że celem sieci neuronowej jest minimalizacja funkcji wartości, ale nie powiedzieliśmy jeszcze wprost jakie typy takiej funkcji istnieją. W uczeniu maszynowym wyróżnia się dwa główne typy zadania. Jest to **klasyfikacja** i **predykcja**. Klasyfikacja polega na przypisaniu przykładu do odpowiedniej klasy, np. jak w przykładzie z kotami do klasy kotów lub braku kotów na zdjęciu. W ten sposób możemy klasyfikować przeróżne rzeczy: chociażby wykrywać twarze lub stwierdzić czy narośle rakowe jest niebezpieczne. Inną możliwością klasyfikacji jest klasyfikacja do wielu klas. W ten sposób możemy klasyfikować np. litery, gatunki zwierząt czy ceny. Aby dokonać klasyfikacji w sieci neuronowej, należy zwrócić liczbę od 0 do 1 w przypadku klasyfikacji dwuklasowej. W przypadku klasyfikacji wieloklasowej należy zwrócić **n** elementową dystrybucję prawdopodobieństwa $[x_1, x_2, \ldots x_n]$, gdzie każdy element oznacza prawdopodobieństwo, że dany przykład należy do danej klasy. Lista ta powinna się sumować do 1. Wyróżniliśmy też inny sposób działania sieci neuronowych i nazwaliśmy go predykcją. Predykcja polega na przewidywaniu jakiejś wartości. Możemy przewidywać np. temperaturę, wyniki wyborów czy wielkość PKB. To wszystko są wartości numeryczne w pewnym przedziale, więc nasza sieć będzie zwracać liczbę naturalną. W tym wypadku wyjściem może być, odpowiednio zeskalowane, normalne wyjście pojedynczego lub kilku neuronów.

Ryc. 3.6: Normalizacja danych



Przed wprowadzeniem danych do sieci neuronowej chcemy je często przetworzyć w jakiś sposób. Działa tutaj zasada mówiąca, że lepsza reprezentacja prowadzi do lepszych rezultatów. Ta maksyma odnosi się zresztą, również do ludzi. Wolelibyśmy, powiedzmy grać w szachy, używając do tego planszy niż mówić na głos nasze ruchy, mimo iż oba rozwiązania opisują tę samą grę. Podobny fenomen możemy obserwować w sieciach neuronowych. One też lepiej radzą sobie z pewnymi rodzajami informacji niż innymi mniej przystosowanymi do bycia wejściem. Dlatego my chcąc osiągnąć jak najlepsze rezultaty, przetwarzamy dane w pewien sposób, licząc na to, że poprawi to wynik działania sieci. Przetwarzanie danych przed użyciem ich w sieci neuronowej często składa się z **normalizacji**. Normalizacja to proces skalowania danych wejściowych. Popatrzmy na prosty przykład:

Zmierzyliśmy temperaturę w przeciągu siedmiu kolejnych dni i otrzymaliśmy w wyniku zbiór danych $\boldsymbol{D}$.

$$D = [1000000, 1000310, 1000070, 999960, 999760, 1000050, 1000540] \tag{3.32}$$

Widzimy, że zmierzone temperatury są bardzo duże. Trenowanie na tym zbiorze będzie ciężkie, ponieważ relatywne różnice między punktami danych są niewielkie, dużo lepiej byłoby odjąć od każdego takiego punktu 1000000, otrzymując nowy zbiór $\boldsymbol{D_s}$.

$$D_s = [0, 310, 70, -40, -240, 50, 540] \tag{3.33}$$

Teraz nawet dla nas dane stały się bardziej przejrzyste. Ten zbiór leży w bardziej naturalnym przedziale i zazwyczaj trenowanie na nim będzie łatwiejsze. System nie będzie się musiał uczyć ignorować dużych wielkości i zwracać uwagę tylko na liczby na niższych miejscach. Zauważmy też, że niektóre dane zmieniły się z pozytywnych wartości na ujemne. Nie będzie to miało jednak efektu na wynik działania sieci, ponieważ informacja o relatywnych różnicach między przykładami została zachowana.

Ponieważ pozyskiwanie dużych zbiorów danych jest często kosztowne, ludzie używają mniejszych zbiorów w pętli, przetwarzając przykłady po kilka razy w trakcie treningu. Ta praktyka, mimo iż może zwiększyć nasze możliwości trenowania sieci, prowadzi też często do pewnych niepożądanych konsekwencji. Znany efekt z tym związany jest nazywany **nadmiernym dopasowaniem** (ang. overfitting), co znaczy, że sieć osiąga dobre wyniki na zbiorze używanym do ćwiczenia, ale słabe w prawdziwym użytkowaniu albo na zbiorze testowym.

Zbiór testowy jest zbiorem, na którym nie trenujemy, po to, aby uzyskać bezstronną ocenę naszego modelu.

Zobaczmy, w jaki sposób może się objawiać nadmierne dopasowanie. Powiedzmy, że mamy pewien zbiór punktów, który może być opisany prostą linią albo trochę lepiej za pomocą skomplikowanego wielomianu. Jeśli do nauczenia tego dopasowania zostało użytych niewiele przykładów, mielibyśmy tendencje, żeby powiedzieć, że wielomian jest prawdopodobnie nadmiernym dopasowaniem, ponieważ nie jest prawdopodobne, by dobrze opisywał przykłady, których nie użyliśmy do jego stworzenia. Jeśli jednak przykładów było więcej, a dane nadal wskazują, że wielomian jest dobrym dopasowaniem do danych, to uzyskaliśmy potwierdzenie i teraz może bardziej sensownym wyborem jest wielomian, niż linia prosta. Nadmierne dopasowanie może się objawiać w sieci neuronowej jako zapamiętywanie przykładów. Mimo iż klasyczna sieć neuronowa nie ma pamięci, to jednak może przechowywać pewne informacje w swoich wagach. Czasami zdarza się, że taka sieć, zamiast np. uczyć się, z jakich części składają się koty, żeby przewidywać czy na zdjęciu jest kot, zapamiętuje konkretne informacje związane z podanym zdjęciem. Później wystarczy, że sieć 'przypomni' sobie, że widziała dane zdjęcie, aby zaklasyfikować je jako zdjęcie zawierające lub niezawierające kota. Dzieje się tak zazwyczaj, gdy sieć, której używamy, jest duża a przykładów, których używamy do trenowania, jest niewiele. W celu zapobiegania zjawisku otrzymywania lepszych rezultatów, ale tylko na papierze, zostały wymyślone metody **regularyzacji**. Jednymi z nich są tak zwane regularyzacja L1 i L2, które dodają do funkcji wartości pewną karę zależną od wielkości wag w naszej sieci. Funkcja wartości zmienia się następująco:

$$min(V_n) = min(V) + |w| \tag{3.34}$$

$$min(V_n) = min(V) + |w|^2 \tag{3.35}$$

Gdzie równanie (3.34) opisuje regularyzację L1, a (3.35) regularyzację L2. Różnicą jak widać, jest potęga, do której podnosimy wagi, jednak w obu przypadkach wartość wag jest wartością bezwzględną.

Taka zmiana w funkcji wartości spowoduje, że nasza sieć nie będzie trenować, tylko aby zwiększać zadaną wartość, co jest dla nas negatywnym skutkiem, ale za to sprawi, że wagi będą pod presją, aby pozostawać mniejsze. Jest to niewątpliwie pozytywny skutek, biorąc pod uwagę eksplodujący gradient. Także ilość połączeń będzie kontrolowana, ze względu na to, że niepotrzebne połączenia będą zmniejszane, bo poprawi to funkcję wartości. Ta mniejsza liczba połączeń, tak jak w naszym przykładzie z dopasowywaniem linii, sprawi, że wytrenowany model będzie prostszy, co zazwyczaj skutkuje lepszą generalizacją.

Idea, która stoi za obecnym dużym sukcesem sieci neuronowych, jest następująca rekomendacja. Jeśli chcemy zwiększyć osiągnięcia sieci neuronowej na danym problemie, to chcemy zwiększyć jej rozmiar, jeśli nie zauważamy poprawy w funkcji wartości podczas treningu. Natomiast jeśli widzimy poprawę podczas treningu, ale nie jest ona widoczna na zbiorze testowym, to powinniśmy zwiększyć wielkość zbioru, na którym trenujemy. Ten wzrost liczby przykładów w zbiorze treningowym jest naturalnym sposobem regularyzacji. Widzimy, że samym zwiększaniem wielkości sieci i ilości danych jesteśmy w stanie poprawić wynik osiągany na naszym problemie. Jest to rzadko spotykana właściwość w świecie informatyki, gdzie zazwyczaj każde zachowanie trzeba zakodować w sposób bezpośredni, gdyż komputery cechują się wielką karnością i ogromnym brakiem wyobraźni. Taki jest klasyczny obraz sytuacji. Jednak sieci neuronowe udowadniają coś przeciwnego. Odpowiednio zaprogramowany komputer może być kreatywny, nie poruszać się utartymi schematami, a nawet uczyć się na własną rękę. Jest to naprawdę wielki przełom, który pozwala nam mieć nadzieję, że będziemy w stanie w przyszłości wykorzystywać do pożytecznych celów moc obliczeniową, która do tej pory rośnie w sposób wykładniczy. Wzrost wykładniczy oznacza że wzrost danej wielkości przyspiesza bardzo szybko. Jest to jeden z niewielu zasobów, które zachowują się w ten sposób. Połączenie tej ogromnej mocy z technikami przetwarzania danych może dawać nam nadzieję na przełom, który może nastąpić w najbliższej przyszłości.

Rozdział 4

Symboliczne AI

Na poprzednich stronach rozmawialiśmy sporo o statystycznym podejściu do sztucznej inteligencji. Istnieje jednak inne podejście nazywane symbolicznym AI. Nie jest ono obecnie tak popularne czy podkreślane w nauce o sztucznej inteligencji, ale historycznie było ważną jej dziedziną. Sądzimy, że obecny ruch oddalający nas od symbolicznych pomysłów wynika częściowo z braku sukcesu w oparciu symboli o sub-symboliczne znaczenie. Ludzie, kiedy mówią o danym pojęciu, przypominają sobie wiele faktów jego dotyczących. Kiedy chociażby mówimy o kocie, to możemy mieć przed oczami jego wygląd, sposób poruszania się, wiemy, jakie jest jego ulubione jedzenie itd. Nie jest więc tak, że pojęcie kota jest niezależne od innych. Raczej należy ono do sieci pojęć, z której każde definiuje inne, z którymi jest połączone, ale jednocześnie uzyskuje swoje znaczenie poprzez inne pojęcia. Tej właściwości często brakuje w symbolicznych systemach. Są one raczej nakierowane na manipulację symbolami w sensie matematycznym. Nacisk rzadko jest położony na rozbudowywanie arsenału możliwości systemu, poprzez dodawanie nowych pojęć. Jednocześnie powiedzieliśmy, że symbole nie mają zazwyczaj połączenia z sub-symbolicznymi systemami takimi jak sieci neuronowe. Sprawia to, że symbole, którymi manipulują te systemy, nie mają dla nich takiego znaczenia jak dla nas. Jest to zupełnie inne podejście. Jednak mimo tych ograniczeń systemy manipulacji symbolami odniosły spory sukces. Podajmy tu przykład programu LT (ang. Logic Theorist) napisanego w 1956 roku przez Newell'a, Simon'a i Shaw'a. LT wykorzystywał strukturę drzewa, na której dokonywał rozumowania, aplikując zmiany do wyrażenia oparte na zasadach logiki i matematyki. Jeśli poprzez te zmiany doszedł on do rozwiązania, to kończył poszukiwania, a ścieżka prowadząca od prepozycji do wyniku była nazywana dowodem. Żeby ograniczyć rozgałęzianie możliwości, LT posiadał pewne heurystyki, które ograniczały przeprowadzanie pewnych operacji, żeby rozmiar drzewa wyszukiwań pozostawał sensowny. Program ten udowodnił 38 z 52 twierdzeń w pewnym rozdziale książki „Principia Mathematica". Udało mu się także znaleźć bardziej elegancki dowód pewnego twierdzenia, jednak nie został on przyjęty do czasopisma matematycznego ze względu na swoją prostotę. Osoba, która oceniała to zgłoszenie, prawdopodobnie nie zauważyła, że autorem był komputer.

Symboliczne systemy definiują pewien zestaw formalnych zasad i używają ich do dochodzenia do pewnych konkluzji. W ten sposób można by rozwiązać na przykład problem różniczkowania albo następujący sylogizm:

Wszyscy ludzie są śmiertelni.

Sokrates jest człowiekiem.

Tu chcielibyśmy, by nasz system skonkludował oczywiste:

Stąd wynika, że Sokrates jest śmiertelny.

Naturalnym rozwinięciem rozumowania symbolicznego są techniki propagacji ograniczeń (ang. constraint propagation), które sprawiają, że wszystkie zdefiniowane ograniczenia są ze sobą zgodne. Na przykład możemy sprawdzić, czy następujące ograniczenia są ze sobą zgodne:

Wszystkie jabłka są czerwone.

Moje jabłko jest zielone.

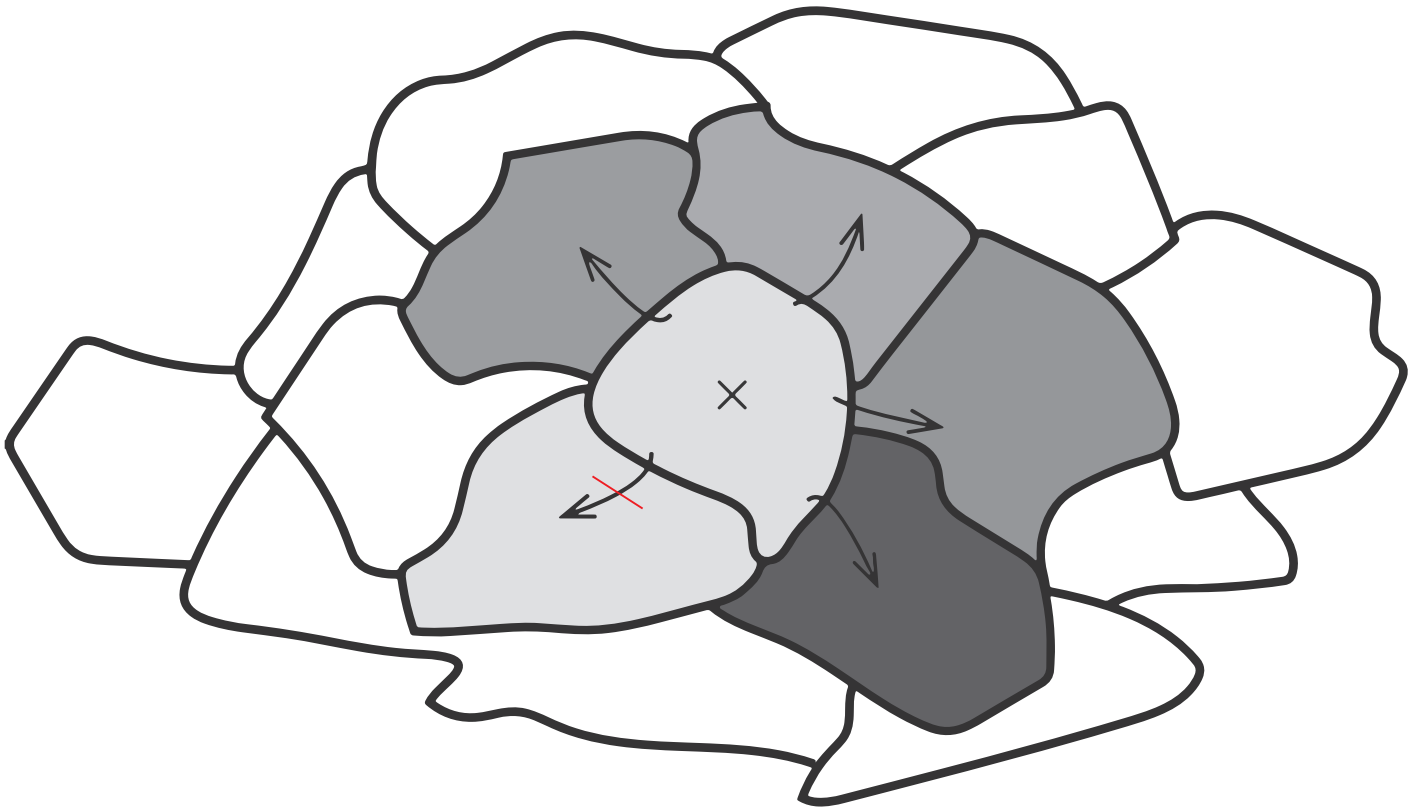
Tu chcemy, aby nasz system skonkludował niemożliwość prawdziwości tych dwóch stwierdzeń w tym samym czasie. Na podstawie podobnych zasad działają tak zwane systemy eksperckie zawdzięczające swoją nazwę od w teorii, naśladowania eksperta. Takie systemy były początkowo najczęściej wykorzystywanymi w przemyśle. Stworzenie takiego systemu wymagało najpierw zebrania specjalistycznej wiedzy od ekspertów. Następnie zapisywało się tę wiedzę przy pomocy różnych zasad, często przy pomocy popularnych w informatyce zasad jeśli-to. Były to często metody nieróżniące się wiele od klasycznych metod programowania. Systemy eksperckie były i są wykorzystywane w różnych specjalistycznych dziedzinach np. prawie, które jest czasami podatne na sformalizowanie w taki sposób ze względu na występowanie w nim wielu zasad.

Jako część metod symbolicznych uznalibyśmy także metody poszukiwania w drzewie. Te techniki dają nam sposób patrzenia na różne wybory, których moglibyśmy dokonać. Wyobraźmy, że znajdujemy się w wielkim zamku z wielkoma pokojami. Chcemy wyjść na zewnątrz, lecz nie znamy drogi, która tam prowadzi. Musimy więc spróbować drzwi, które prowadzą z naszego pokoju do innego, w którym może być wyjście. Jeśli z naszego pokoju wychodzi kilka różnych drzwi, to musimy mieć jakiś sposób dokonania wyboru pomiędzy nimi. W następnym pokoju napotkamy prawdopodobnie podobny problem, gdzie będziemy musieli wybrać między wieloma drzwiami. Metody poszukiwania w drzewie dają nam sposób na śledzenie odwiedzonych pokoi. Zbierając te informacje, dają nam one możliwość dokonywania najlepszej decyzji w danych okolicznościach.

4.1 Rozwiązywanie poprzez wyszukiwanie

Widzieliśmy już jedno rozwiązanie problemu poprzez wyszukiwanie. Czy pamiętasz, co to było? Kiedy szukaliśmy sposobu, aby znaleźć minimum funkcji wartości, zaproponowaliśmy na początku, aby sprawdzać możliwe rozwiązania w sposób losowy przez jakiś czas, a następnie wybrać najlepsze odwiedzone rozwiązanie. To jest właśnie rozwiązanie przez wyszukiwanie. Jednym z problemów, które moglibyśmy rozwiązać poprzez wyszukiwanie, jest problem **propagacji ograniczeń**. W takim problemie staramy się sprawdzić, czy wszystkie nałożone ograniczenia są ze sobą zgodne, czy mówiąc inaczej, nie zaprzeczają sobie. Na przykład w problemie kolorowania map, pytamy się, czy i w jaki sposób jesteśmy w stanie pokolorować daną mapę za pomocą $\boldsymbol{n}$ kolorów, w ten sposób, aby przylegające kraje były pokryte innymi kolorami tuszu. Dodatkowe pytanie, jakie możemy zadać to: czy istnieje, a jeśli tak to, jaka jest najmniejsza liczba kolorów, za pomocą której możemy pokolorować dowolną mapę? Na to pytanie odpowiada twierdzenie o czterech kolorach. Mówi nam ono, że za pomocą czterech kolorów jesteśmy w stanie pokolorować dowolną mapę. Taką hipotezę postawiono już w XIX w. ale na rozwiązanie tego problemu przyszło nam poczekać aż do roku 1976, w którym przy użyciu komputera sprawdzono 1936 przypadków możliwych ułożeń mapy. Jednak później powstały pewne wątpliwości co do poprawności rozwiązania, które jednak udało się rozwiać znów przy pomocy komputerowego wspomagania. Jest to jeden z ciekawszych dowodów w matematyce, ponieważ ciężko jest go sprawdzić człowiekowi i pokazuje nam, że nowy rodzaj dowodów jest możliwy. Wracając jednak do głównego tematu, przypomnijmy, że chcemy rozwiązać ten problem dla konkretnego przypadku, więc nie wystarczy nam dowód, ale potrzebujemy też konkretnych kolorów dla danych państw. Moglibyśmy w teorii rozwiązać ten problem metodą prób i błędów, ale istnieje dużo szybszy sposób. Używa on idei spójności. Kolor państwa jest spójny w węźle (ang. node-consistent), jeśli jest spójny ze samym sobą. Spójność w węźle jest w oczywisty sposób zachowana, jeśli tylko używamy legalnego koloru. Kolor będzie spójny w łuku (ang. arc-consistent), jeśli jest spójny ze samym sobą oraz z kolorami przylegających państw. Lepszym rozwiązaniem problemu kolorowania mapy jest wybranie jakiegoś koloru dla jednego państwa i sprawdzenie spójności w łuku. Następnie, jeśli rozwiązanie jest spójne, wybierzemy kolor dla innego państwa i sprawdzimy jego spójność w łuku. Postępujemy w ten sposób, tak długo aż nie rozwiązaliśmy problemu lub problem stał się niespójny. W takim przypadku cofniemy się do poprzedniego rozwiązania, które było spójne w łuku i spróbujemy innych kolorów. To cofanie nazywane jest **nawrotem** (ang. backtracking). Nawroty są dość częstym pomysłem wykorzystywanym w drzewie poszukiwań, o którym powiemy dalej. Pozwalają nam one cofnąć się do poprzednio odwiedzonego stanu i wybrać inną ścieżkę, czy to z powodu bycia niespójnym, czy też, aby sprawdzić inne możliwości. Pomysł cofania pewnych informacji w górę drzewa będzie wykorzystywany dalej w drzewach poszukiwań.

Ryc. 4.1: Propagacja ograniczeń



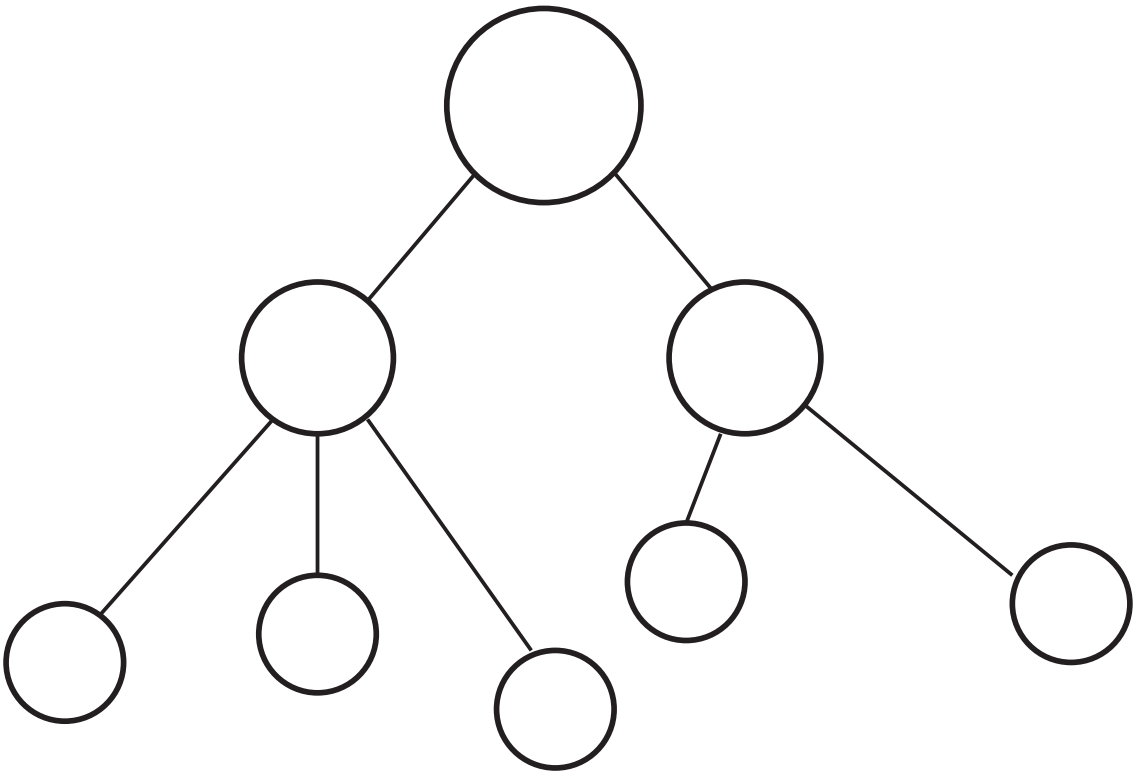
4.2 Drzewa poszukiwań

Drzewo jest strukturą danych, która bierze swoją nazwę z podobieństwa do drzewa. Drzewa wyrastają z korzenia, a gałęzie rozgałęziają się na coraz to mniejsze. W informatyce drzewa umieszczamy na głowie, tak że korzeń znajduje się u góry a czubek drzewa wraz z większością gałęzi na samym dole. Ten sposób rysunku sprzyja naturalnemu rozwijaniu drzew od góry do dołu, tak samo, jak podczas pisania. Drzewo jako struktura danych składa się z węzłów połączonych krawędziami. Każdy węzeł oznacza pewien stan, a krawędź istnienie pewnej relacji pomiędzy tymi stanami. Stanami może być np. numer pokoju, w którym się znajdujemy a krawędziami drzwi, które łączą te pokoje. Każdy węzeł, z którego wyrasta gałąź, nazywany jest węzłem nadrzędnym (ang. parent node), a każdy węzeł posiadający węzeł nadrzędny nazywany jest węzłem podrzędnym (ang. child node).

Drzewo poszukiwań jest techniką, której nazwa nawiązuje do wykonywania poszukiwań na strukturze danych, jakiej jest drzewo. Opisaliśmy już jedną metaforę je opisującą, kiedy to wybierając odnogę drzewa, wybieramy kolejne drzwi. Jakiego rodzaju problemy mogłyby być dobrze opisane jako takie drzewo poszukiwań? Wygląda na to, że każdy problem, w ciągu rozwiązywania którego, wielokrotnie podejmujemy podobną decyzję, byłby podatny takiemu podejściu. Na przykład: wyszukiwanie drogi w labiryncie, planowanie diety czy granie w szachy. Wszystkie te problemy wydają się dobrze zdefiniowane dla algorytmów opierających się na drzewa. Jeśli nie rozumiesz dlaczego, to zastanówmy się nad tym problemem. Po pierwsze, aby nasze dane mogły być dobrze opisane przez drzewo, to każdy stan musi mieć ze sobą coś wspólnego, ale również każdy stan powinien być rozróżnialny od innych. I tak w labiryncie możemy zapisywać unikalne położenie, przy planowaniu diety ilość kalorii a w grze w szachy pozycję. Następnie wszystkie połączenia między stanami powinny być tego samego rodzaju. Wybieranie między naszyjnikiem a ilością kalorii nie ma większego sensu. Natomiast wybór między sałatką a burgerem jest bardziej rozsądny. Tak więc podobny rodzaj połączeń powinien występować między wszystkimi stanami, aby móc dokonać między nimi wyboru. W pewnym sensie możemy myśleć o rozgałęzianiu się drzewa, od jego korzenia, jak o różnych możliwych historiach, które się nam prezentują. Jeśli wybralibyśmy pewną odnogę drzewa i podejmowali decyzje, żeby ją osiągnąć, to ona stałaby się prawdą, a więc historią wydarzeń. Jeśli jednak podjęlibyśmy decyzje prowadzące do innego miejsca, to ono byłoby historią, która się wydarzyła. Istnieje tu pewne podobieństwo do interpretacji wielu światów w mechanice kwantowej, która mówi, że na poziomie kwantowym świat rozszczepia się na różne historie, które stają się nowymi wszechświatami. Różnica jest taka, że w mechanice kwantowej wybór jest dokonywany przez losowe zachowanie cząsteczki a w drzewach poszukiwań przez maksymalizującego cel aktora. Możemy o drzewie poszukiwań myśleć jak o wybieraniu takiego

kwantowego wszechświata. Sam wybór możemy nawet zobrazować jak przechodzenie przez drzwi między kolejnymi wszechświatami. Ta metafora pomoże nam dalej w interpretacji bardziej abstrakcyjnych pomysłów związanych z drzewami poszukiwań. Wspomnieliśmy także o jednej z metod wykorzystywanych w drzewach poszukiwań, jaką jest nawrot. Pomyślmy teraz, co oznacza on w metaforze wielu światów. Jeśli odwiedzimy pewien stan i się z niego cofniemy do innego poprzedzającego stanu, to znaczy, że naprawdę go nie odwiedziliśmy. Jeśli rzeczywiście byśmy w nim byli to znaczy, że nie moglibyśmy się cofnąć do poprzedzającego stanu, ponieważ czas płynie tylko w jedną stronę. Jednak w nawrocie cofamy się do poprzedzającego stanu. Oznacza to, że nie byliśmy w tym stanie, a więc było to tylko nasze wyobrażenie bycia w danym stanie. Inaczej mówiąc, przeprowadziliśmy symulację. Symulacje są jednym z ulubionych narzędzi w skrzynce kogoś zajmującego się sztuczną inteligencją. Co oznacza słowo symulacja? Symulacja jest pewnym uproszczeniem rzeczywistości, które jednak zachowuje pewne ważne jego elementy w celu przewidywania przyszłości. Próbujemy symulować zachowania, żeby podejmować najlepsze decyzje. Ludzie tworzą przeróżne symulacje, zaczynając od ekonomicznych i finansowych, symulacji zachowania ludzi i tłumów, symulacje działania pojazdów, symulacje robotów. Ludzie programujący rozwiązania sztucznej inteligencji lubią symulacje, ponieważ pozwalają im one na sprawdzanie swoich systemów w środowisku testowym. Symulowana rzeczywistość może nam też dostarczyć danych do trenowania naszych systemów takich jak sieci neuronowe. Dużo ciężej jest wpuścić system sztucznej inteligencji do prawdziwego świata niż do symulacji, co wynika z zagrożeń z tym związanych, możliwości testowania systemu, jak już wcześniej wspomnieliśmy, i sposobów komunikacji takiego aktora z rzeczywistością. Czasami, żeby działać w rzeczywistości, konieczny byłby robot, w przeciwieństwie do symulacji. Dlatego właśnie naukowcy wolą zazwyczaj używać do badań środowisk testowych, jakimi są symulacje. Najprostsze takie modele rzeczywistości powstały dawno temu na bliskim wschodzie i gdzieś w Azji. Mowa tu oczywiście o grach planszowych takich jak szachy, Go, shogi. Oddają one pewien aspekt rzeczywistości w uproszczonej formie. Chociażby szachy były symulacją kierowania polem walki. Takie proste gry są nam dzisiaj bardzo przydatne do rozwijania rozwiązań sztucznej inteligencji. Właśnie na nich z sukcesem zastosowano techniki drzew poszukiwań, i to one stoją za niektórymi najnowszymi postępami w dziedzinie, służąc jako środowisko testowe.

Ryc. 4.2: Drzewo poszukiwań

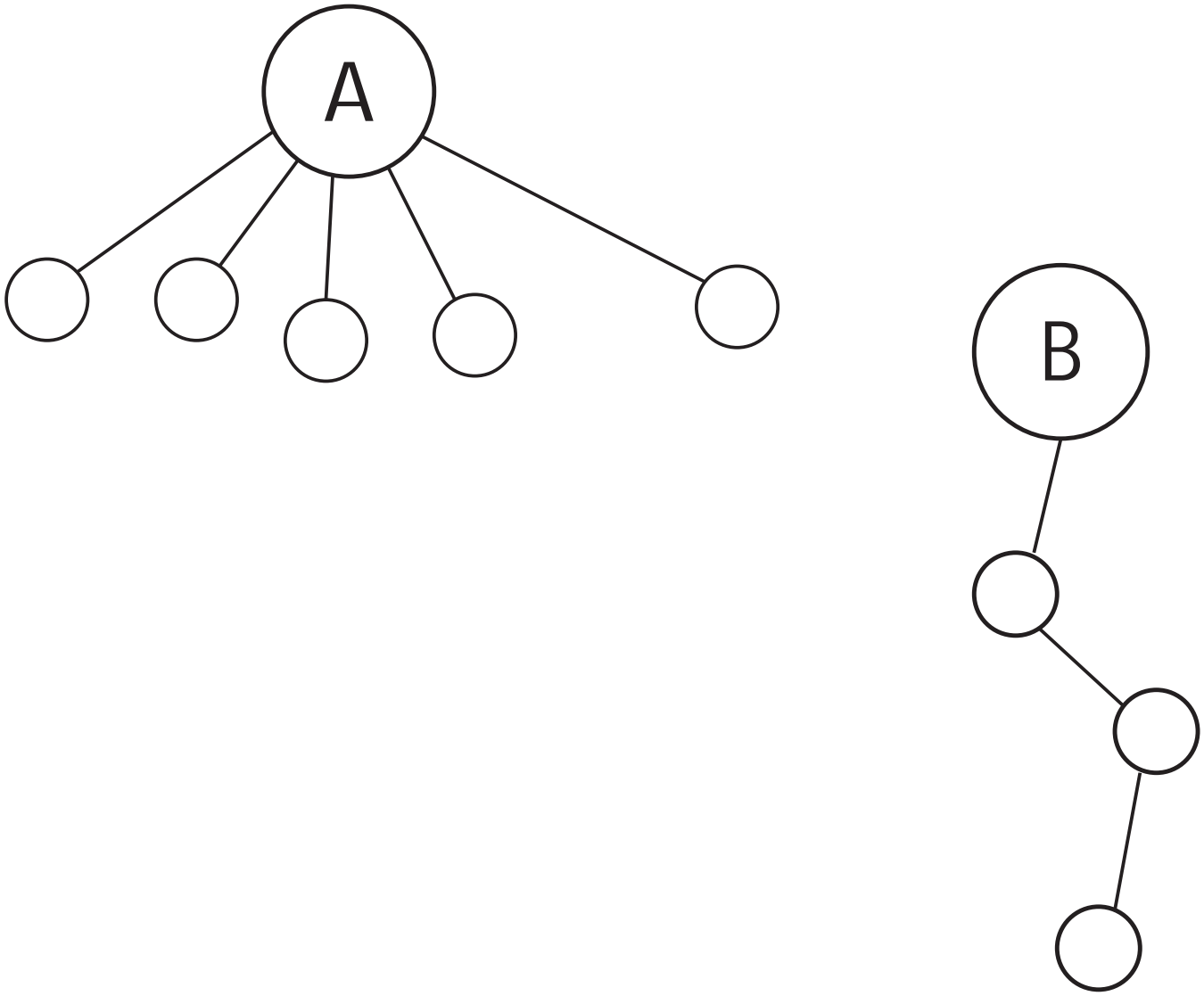


4.3 Poszukiwanie na głębokość i rozpiętość

Powinniśmy teraz skupić się na problemie kolejności, w której będziemy sprawdzać możliwe przyszłości. Możemy o tym myśleć jak o planie jak otwierać kolejne drzwi. Zastanówmy się jakie podstawowe właściwości ma drzewo poszukiwań. Patrząc się, nawet wizualnie, możemy wyróżnić dwie główne cechy węzłów w drzewie. Jak myślisz jakie są to cechy? Jednymi z ważniejszych cech jest to, jak głęboko węzeł znajduje się względem korzenia i ilość rozgałęzień, na które się rozgałęzia. Na tych dwóch cechach skupiają się algorytmy poszukiwania na głębokość i na rozpiętość. Jednym sposobem na eksplorowanie takiego drzewa poszukiwań jest patrzeć się tak głęboko, jak to jest możliwe, nie zwracając uwagi na inne drogi, które moglibyśmy obrać. To podejście ma nazwę **poszukiwania w głąb** (ang. depth first). Algorytm poszukiwania w głąb zawsze na początku będzie się starał patrzeć, jak najgłębiej jest to możliwe, wybierając do sprawdzenia w następnej kolejności zawsze węzeł podrzędny, jeśli jest to możliwe. Kiedy osiągamy węzeł końcowy, musimy się nawrócić do ostatniego miejsca, w którym mieliśmy wybór i stamtąd rozpocząć nasze poszukiwania, w podobny sposób, nie odwiedzając już jednak raz odwiedzonych węzłów, jeśli nie jest to konieczne. Jeśli wrócimy do naszej metafory o wybieraniu drzwi w zamku, odpowiadałoby to wybieranie pierwszych napotkanych drzwi w pokoju i przechodzenie dalej do momentu aż znajdziemy wyjście albo napotkamy na pokój bez drzwi. Kiedy dojdziemy do takiego pokoju, to zawracamy do poprzedniego pokoju, w którym byliśmy i kontynuujemy poszukiwania w ten sam sposób. Jakie mocne strony ma ten algorytm? Jedną silną stroną tego podejścia jest to, że nie musimy być bardzo dobrzy w ocenianiu sytuacji, w której jesteśmy, ponieważ dostajemy się do logicznego końca możliwości i to tam możemy dokonać oceny. Jedną słabą stroną poszukiwania w głąb jest to, że nie sprawdza ono wielu możliwości, tylko ślepo wybiera jedną drogę, którą podąża. Zupełnie inną możliwością jest **poszukiwanie wszerz**. Analogicznie do poszukiwania w głąb, poszukiwanie wszerz będzie najpierw sprawdzać wszystkie możliwości na jednym poziomie, zanim będzie poruszać się w głąb. Zużywa ono wszystkie węzły podrzędne przed przejściem do węzłów podrzędnych względem podrzędnych. W naszej metaforze zamku musimy najpierw sprawdzić wszystkie drzwi prowadzące z naszego pokoju, później sprawdzamy wszystkie drzwi w pokojach, do których prowadzą drzwi z naszego pokoju itd. Żeby to wszystko ogarnąć, musielibyśmy stworzyć mapę i zapisywać na niej wszystkie połączenia. Podobnie działają oba z tych algorytmów. Zapisują odwiedzone stany w strukturze drzewa. Silne i słabe strony algorytmu poszukiwania wszerz będą zamienione z poszukiwaniem w głąb. To podejście bierze pod uwagę wszystkie możliwe wybory przed przejściem w głąb, a więc sprawdza wszystkie możliwości, które napotyka. Wiąże się to jednak, z tym że może nie dojść tak głęboko, jak poszukiwanie w głąb, jeśli nie otrzyma wystarczającego czasu, więc system oceniający sytuację musi być dobry, ponieważ tylko kilka ruchów w głąb, gdzie będzie trzeba

dokonać wyboru, może być niejasnym czy sytuacja jest korzystna. Obie te metody mimo swoich różnic są w pewien sposób do siebie podobne. Stanowią, jak można by powiedzieć, swoje lustrzane odbicie. Są jednymi z najprostszych metod przeszukiwania drzewa. Obrazują jednocześnie kluczowy problem, który napotykamy podczas wyboru następnego węzła, który chcemy odwiedzić. Czy należy sprawdzić wiele alternatyw, czy lepiej poruszać się głębiej? Ten problem będzie dla nas dalej istotny i algorytmy dalej zaprezentowane mają unikalne metody radzenia sobie z nimi. Obie z tych metod utrzymują również w pamięci najlepszy do tej pory odwiedzony węzeł, aby zwrócić go jako odpowiedź. Kiedy odwiedzamy nowy węzeł, zawsze sprawdzamy, czy nie jest lepszy od do tej pory najlepszego i jeśli tak to zamieniamy najlepszy węzeł na obecnie odwiedzany. Ta właściwość wymaga funkcji wartości, która będzie w stanie ocenić każdy odwiedzony stan. W algorytmach tu zaprezentowanych funkcja wartości jest używana dopiero na końcu po fazie rozwijania węzłów. Węzły są najpierw rozwijane, a następnie ocena przez funkcję wartości jest stosowana na węzłach liściach, czyli takich, które nie mają żadnych podrzędnych węzłów. Spośród węzłów liści wybierana jest najlepsza gałąź, a następnie cofamy się do początku tej gałęzi i dokonujemy pierwszego wyboru, który prowadzi do tej najkorzystniejszej historii. W następnym ruchu zazwyczaj wszystko jest resetowane i poszukiwanie rozpoczyna się na nowo. Ważne jest tu także to, że sposób symulacji następnych stanów musi być dobry przy dużej ilości rozgałęzień, tak żeby ocena sytuacji w węźle liściu nie odbiegała zbyt dużo od poprawnej oceny naszej sytuacji z powodu różnic między zasymulowanym a prawdziwym rozwinięciem. Dlatego tego typu algorytmy nadają się najlepiej do sytuacji, w których symulacja jest idealna takich jak np. szachy. W szachach możemy dokładnie powiedzieć jakie możliwości istnieją w danej sytuacji i do czego prowadzą, dlatego będziemy dalej wykorzystywać ten przykład.

Ryc. 4.3: Poszukiwanie na rozpiętość i głębokość



4.4 Minimax

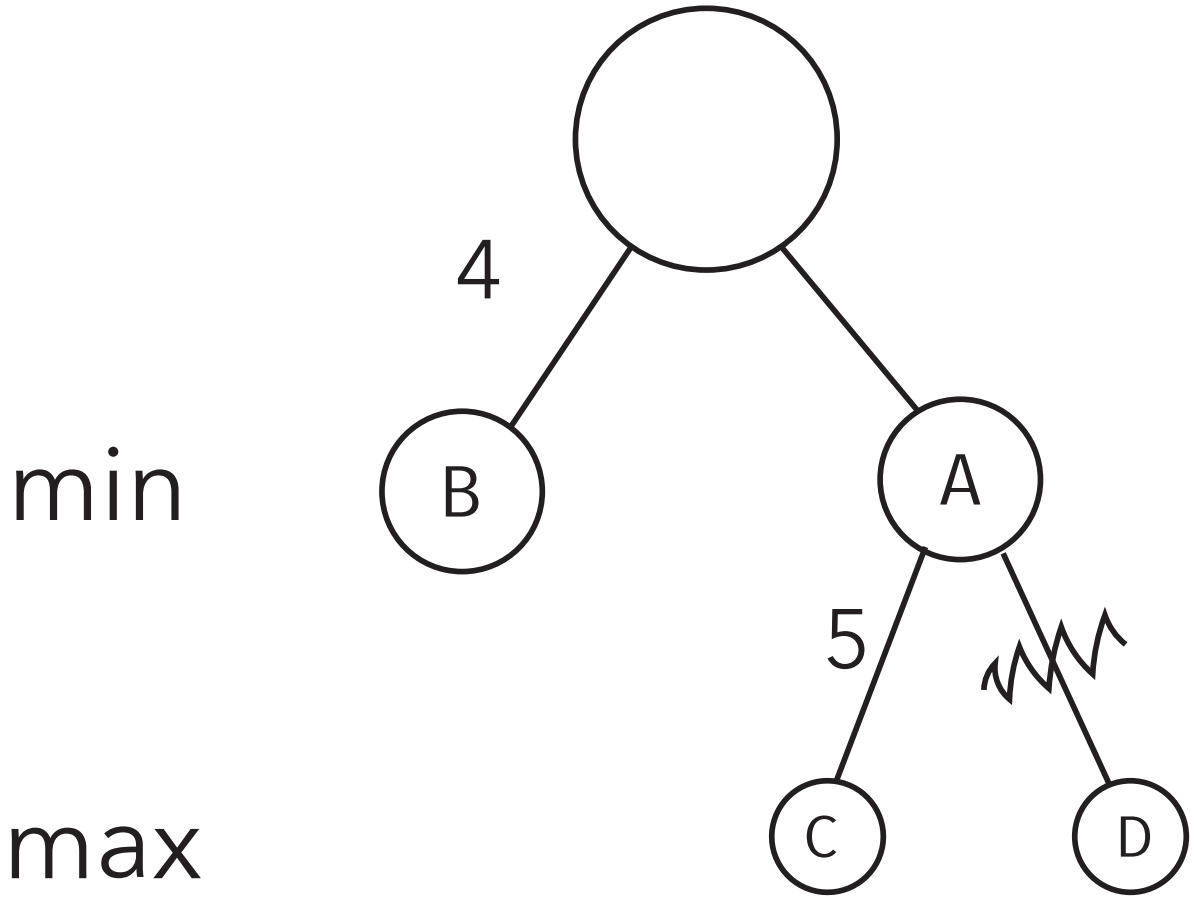
Mówiliśmy o tym, że możemy próbować rozwiązać gry dla dwóch graczy, takie jak warcaby czy szachy używając technik wyszukiwania. Czy to znaczy, że możemy po prostu wziąć algorytm poszukiwania w głąb lub wszerz i wykorzystać go do znalezienia najlepszego ruchu? Używając tamtych algorytmów, musieliśmy podać funkcję wartości i wybieraliśmy ruch ze względu na jej maksymalizację, ale na przykład w szachach, ta sama pozycja może mieć zupełne inną wartość, w zależności od tego, czyj ruch jest następny. Tak dzieje się, ponieważ jeden z graczy chce maksymalizować a drugi minimalizować tę samą funkcję wartości. Załóżmy, że wygrana białego to +1 a wygrana czarnego to -1, remis oznaczmy jako 0. W takiej sytuacji biały chce, żeby funkcja wartości była jak najbliżej 1, bo to daje mu największe szanse na wygraną, natomiast czarny przeciwnie, chce żeby funkcja wartości była jak najniższa i znajdowała się jak najbliżej -1.

$$v_{hat} = max[min[v(a_a, a_b)]] \tag{4.1}$$

Gdzie $\boldsymbol{v_{hat}}$ oznacza funkcję wartości dla jednego z graczy po ruchu przeciwnika i własnym, $\boldsymbol{v}$ oznacza funkcję wartości z jednej perspektywy w zależności od ruchów $\boldsymbol{a_a}$ własnym i $\boldsymbol{a_b}$ oponenta.

Widzimy, że zachodzi tu rekursywna zależność, gdzie jeden z graczy chce minimalizować funkcję a drugi ją maksymalizować, i tak przy każdym ruchu. Nie możemy więc patrzeć tylko na nasze własne ruchy, bo przeciwnik będzie się starał znaleźć luki w naszym rozumowaniu. Musimy patrzeć się na przemian na ruchy dla nas i dla naszego rywala. Chociaż chcemy wybrać najlepszy ruch, to musimy go wybrać z perspektywy najgorszej możliwej sytuacji, którą wybierze dla nas nasz przeciwnik. Jednakże on ma ten sam problem, musi wybrać najgorszy ruch dla nas, wybierając wśród najlepszych ruchów, które zrobimy i tak dalej.

Рис. 4.4: Minimax



$$v_{hat} = max[min[max[min[\dots]]]] \tag{4.2}$$

Możemy zobaczyć, że występuje tu rekursywność, w której etap maksymalizacji następuje po etapie minimalizacji, a po etapie minimalizacji następuje etap maksymalizacji itd. Ta seria wydarzeń prowadzi nas bezpośrednio do pomysłu na algorytm, który wykorzystuje tę zależność. Ten algorytm patrzący się kolejno i próbujący maksymalizować funkcję wartości dla jednego poziomu węzłów i minimalizować dla kolejnego nazywa się **minimax**. Powtórzmy jeszcze raz, jak wygląda jego działanie na przykładzie szachów. Kiedy następuje nasza tura, najpierw określamy wszystkie ruchy, które możemy wykonać i zapisujemy je w drzewie, następnie przechodzimy do powstałych sytuacji i robimy to samo tym razem dla ruchów przeciwnika. Kiedy dojdziemy do pozycji końcowej, np. 'mata' to określamy, kto wygrał i zaprzestajemy poszukiwania. Jednak prawie nigdy nie korzysta się w rzeczywistości z wyszukiwania do końca ze względu na astronomicznie rosnącą liczbę rozgałęzień. Na ogół w pewnym momencie musimy przerwać poszukiwanie i określić wartość niedokończonej sytuacji w węźle za pomocą chociażby sieci neuronowej. Kiedy określimy która gałąź jest najkorzystniejsza, to wybieramy ją i cofamy się do pierwszego ruchu w tej sekwencji. Wykonujemy znaleziony ruch. Czasami w algorytmie minimax nastąpi sytuacja, w której inna gałąź jest wybrana ponad gałąź, którą eksplorujemy i zobaczenie jakiejkolwiek nowej wartości, jakkolwiek dobrej dla nas, nie zmieni ruch, który wykona nasz przeciwnik. Właściwie im lepszą możliwość znajdziemy, tym bardziej przeciwnik nie będzie chciał tam iść w ruchu poprzedzającym, tak więc znalezienie lepszej dla nas sytuacji, gdy jesteśmy przekonani, że przeciwnik i tak nie wybierze tego rozgałęzienia, ze względu na to, że nie jest dla niego dobre jeszcze przed sprawdzeniem nowego rozgałęzienia, nie ma sensu. W takiej sytuacji możliwym jest, aby dokonać płytkiego lub głębokiego odcięcia gałęzi, to znaczy, że nie będziemy takiej gałęzi więcej przeszukiwać. Ta technika nazywana jest odcięciem **alfa-beta** (ang. alpha-beta pruning) i jest często używanym dodatkiem do techniki minimax. Powinna ona być właściwie używana zawsze, gdy mamy taką możliwość, ponieważ ucina ona tylko gałęzie, które nie mogą mieć wpływu na wynik wyszukiwania. Tak więc wyszukiwanie alfa-beta jest równoważne w wyniku do wyszukiwania minimax.

Deep Blue, które pokonało Kasparova w słynnej serii meczów, używało algorytmu alfa-beta, który działał w paraleliźmie na wielu specjalnie do tego przygotowanych procesorach, używał skonstruowanej na te potrzeby funkcji ewaluacji, której parametry były nauczone przez system. Co ciekawe Deep Blue korzystało z wersji algorytmu zwanej **iteracyjne pogłębianie** (ang. iterative deepening) co oznacza, że wyszukiwanie jest ograniczone do pewnego poziomu głębokości i po jego osiągnięciu najlepszy do tej pory wynik zostaje zwrócony, natomiast jeśli pozostało jeszcze czasu na obliczenia, to wybierany jest pewien głębszy poziom graniczny, do którego prowadzi się poszukiwanie.

Ten proces powtarzany jest wielokrotnie w ciągu wyszukiwania jednego ruchu i najlepszy ruch jest zmieniany na ten osiągnięty najgłębszym do tej pory wyszukiwaniem w miarę zwiększania głębokości. Trzeba zauważyć, że rezultaty poprzedniego wyszukiwania zostają utracone w kolejnym wyszukiwaniu, które prowadzi się od nowa. Może ci się to wydawać dużym marnotrawstwem, ale należy pamiętać, że głębsze poziomy zawierają eksponencjalne więcej węzłów niż te bliżej korzenia, więc straty osiągane przy każdym wyszukiwaniu maleją eksponencjalne w porównaniu do wyszukiwania z maksymalnym poziomem głębokości. Co daje nam ten algorytm? Zauważyliśmy, że w czasie jednego wyszukiwania kilka ruchów będzie zwróconych. Teraz, jeśli okazuje się, że nie wiemy dokładnie, ile ma trwać wyszukiwanie, to dzięki tej metodzie będziemy mieć gotowy dobry rezultat w trakcie całego trwania wyszukiwania.

4.5 A*

A* (czytaj z ang. A star) jest algorytmem **najpierw najlepsze** (ang. best first) co znaczy, że wyszukuje na początku te węzły, które podejrzewa, że będą najlepsze. Żeby to zrobić, potrzebuje pewnego sposobu estymacji funkcji wartości. Gałęzią sprawdzaną przez A* będzie ta, która minimalizuje koszt dotarcia do węzła i spodziewany koszt od tego węzła.

$$min[v(n)] = min[g(n) + h(n)] \tag{4.3}$$

Gdzie $\boldsymbol{v}$ jest funkcją wartości, $\boldsymbol{g}$ jest kosztem dotarcia do węzła, a $\boldsymbol{h}$ jest spodziewanym kosztem od tego węzła do celu.

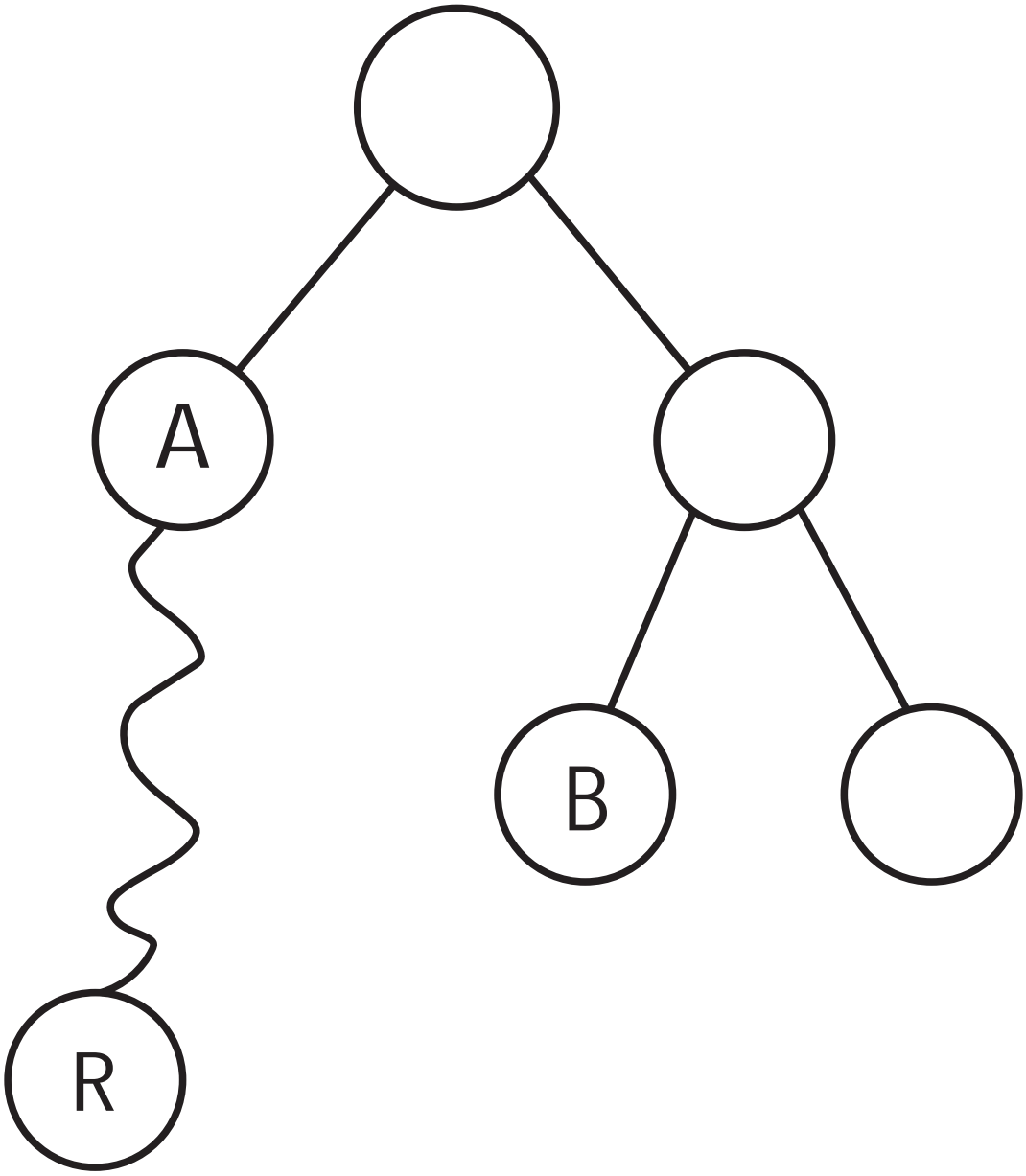
Najprostszym przykładem sytuacji, w której A* byłby użyteczny, jest problem poszukiwania najlepszej ścieżki. Powiedzmy na przykład, że chcemy znaleźć drogę prowadzącą z miasta A do miasta B, która prowadzi przez pewien zbiór miast C. Może poruszamy się z miasta A do B często, więc chcemy znaleźć najlepszą ścieżkę, która prowadzi z A do B. Nie znamy jednak dokładnych odległości między miastami. W tym przypadku możemy użyć algorytmu A*, gdzie $\boldsymbol{g}$ byłoby realnym czasem drogi, którą pokonaliśmy przeliczonym na dystans, a $\boldsymbol{h}$ odległością pomiędzy miastami na mapie, spośród których próbujemy wybrać najkorzystniejsze. Widzimy więc, że część funkcji wartości zależy od rzeczywiście przebytego dystansu, a druga część od spodziewanego dystansu do przebycia. Naprawdę niekoniecznie musi to być dystans przebyty, a mogą być to wielkości na mapie. Kiedy np. program próbuje znaleźć najlepszą trasę dla naszego samochodu, wykorzystuje rozwiązania podobne do algorytmu A*. Sprawdza, ile wynosi odległość w linii prostej do celu w kilku punktach, do których możemy pojechać i dodaje do niej odległość rzeczywistej drogi prowadzącej do tych punktów, która jest do pokonania. Następnie przesuwa się do tych punktów i wykonuje podobną procedurę aż do znalezienia najlepszej trasy. Algorytm A* ma tę korzystną właściwość, że

odległość spodziewana $\boldsymbol{h}$ jest zawsze mniejsza niż rzeczywista odległość, którą pokonamy. Pozwala to na określenie właściwości, która mówi, że koszt dotarcia do sąsiedniego następującego węzła i podróż od niego do celu jest zawsze nie większa niż bezpośredni koszt podróży do celu z naszego węzła. Może się to wydawać oczywiste, ale nie musi być zawsze zapewnione dla generalnego przypadku wyszukiwania na grafie. Algorytm A* został po raz pierwszy użyty w robocie Shakey, który był robotem poruszającym się samodzielnie na kołach pomiędzy pewnymi miejscami. Potrzebował więc sposobu planowania swojej trasy. Shakey miał dostawać zadanie i być w stanie rozbić je na mniejsze części, które będą konieczne do realizacji celu. Shakey do realizowania tego celu posiadał kilka umiejętności. Potrafił podróżować z jednej lokacji do drugiej (dzięki algorytmowi A*), otwierać drzwi, włączać światła, popychać obiekty. Robot posiadał system wizji, dzięki któremu orientował się w otoczeniu. Te umiejętności pozwalały mu na wykonywanie prostych zadań. Shakey był jednym z pierwszych projektów integrujących robotykę i AI, używając wiedzy z obu dziedzin do stworzenia autonomicznego robota.

4.6 MCTS

Drzewo poszukiwań Monte Carlo (ang. Monte Carlo tree search) w skórcie MCTS, jest metodą, którą polega w mocny sposób na losowości, stąd Monte Carlo w nazwie. MCTS nie potrzebuje funkcji wartości, ponieważ dokonuje tzw. **rozwinięć** (ang. rollout) co oznacza, że zostaje przeprowadzona symulacja od liścia do węzła końcowego, czyli takiego, który nie ma żadnych węzłów podrzędnych. Symulacja odbywa się poprzez wybranie losowej akcji i użycie jej i modelu świata do wygenerowania następnej sytuacji. Powtarzamy tę czynność, aż nie dojdziemy do węzła końcowego. W tym przypadku musimy co prawda użyć funkcji wartości do określenia sytuacji końcowej i nawrócenia wartości do liścia, jednak w grach, w których ten algorytm jest często stosowany jest inaczej. Na końcu rozgrywki funkcja wartości jest powszechnie znana, ponieważ jest to rezultat gry, a on musi być jasno określony w zasadach. Nie musimy wiedzieć nic więcej poza zasadami takiej gry. Tak więc np. w przypadku szachów dokonujemy najpierw losowego posunięcia dla nas, powiedzmy, poruszamy skoczkiem, następnie wykonujemy posunięcie dla przeciwnika, powiedzmy, ruszamy pionkiem. Kontynuujemy tak na przemian, aż nie dojdziemy przypadkowo do sytuacji, w której gra jest matem, patem lub remisem. Wtedy określamy wartość dla danego węzła końcowego. Dla szachów może to być +1, 0 lub -1. Każdy węzeł utrzymuje informacje o dwóch zmiennych, sumie wartości węzłów podrzędnych $\boldsymbol{v}$ oraz liczbie wizyt węzła $\boldsymbol{n}$. Kiedy rozwinięcie jest przeprowadzane to wartości $\boldsymbol{v}$ i $\boldsymbol{n}$ są nawracane do węzłów nadrzędnych, a stamtąd do węzłów nadrzędnych do tych nadrzędnych itd. Na każdym nadrzędnym poziomie wartości $\boldsymbol{v}$ i $\boldsymbol{n}$ są zmieniane poprzez dodanie następujących wartości:

Рис. 4.5: MCTS



$$\delta_v = \text{value of rollout} \tag{4.4}$$

$$\delta_n = 1 \tag{4.5}$$

Gdzie $\boldsymbol{v}$ zmienia się o wartość rozwinięcia, a $\boldsymbol{n}$ jest zwiększane o 1, co znaczy, że jedna dodatkowa symulacja została przeprowadzona.

Kiedy rozwinięcie zostaje przeprowadzone to cała ścieżka, która nas do tego miejsca doprowadziła, zostaje wykasowana z pamięci, a pozostają jedynie informacje o $\boldsymbol{\delta_v}$ i $\boldsymbol{\delta_n}$. Te informacje są używane do odpowiedniej zmiany wartości węzłów nadrzędnych. Na razie nie powiedzieliśmy, skąd te węzły się biorą i pozostawimy tę informację na koniec. Teraz musisz jedynie wiedzieć, że tak jak w innych algorytmach poszukiwania te węzły istnieją i są czymś innym niż rozwinięcie. Rozwinięcie jest jedynie swego rodzaju próbkowaniem rzeczywistości. Wszystko, co jest konieczne do takiego próbkowania, zostaje usunięte z pamięci, a zostają pozostawione tylko węzły, z których przeprowadzamy to próbkowanie. Żeby wybrać kolejny węzeł, z którego chcemy przeprowadzić rozwinięcie, potrzebujemy jakiegoś równania, które określi nam wartość rozwinięcia z danego węzła. Co możemy zawrzeć w takim równaniu? Po pierwsze chcielibyśmy, aby częściej wizytowane były węzły, które są lepsze, czyli mają większą wartość $\boldsymbol{v}$, ponieważ chcemy sprawdzić czy są rzeczywiście tak dobre jak to ustaliliśmy. Musimy jednak pamiętać, że $\boldsymbol{v}$ będzie zależeć od wartości $\boldsymbol{n}$ czyli ilości takich rozwinięć. Tak więc dobrze byłoby, gdyby to równanie zależało od $\boldsymbol{v/n}$. Taka jest też pierwsza część równania. Jeśli jednak zawarlibyśmy tylko te informacje, to 'dobre' węzły byłyby jedynymi odwiedzanymi. Może się jednak zdarzyć tak, że jedna symulacja, jedno rozwinięcie jest złe, ale wszystkie inne wypadłyby dobrze. Z funkcją wybierającą węzły $\boldsymbol{v/n}$ do rozwinięcia nie jest możliwe, żeby sprawdzić węzeł, któremu się nie poszczęściło. Tak więc dodana jest druga część tego równania, która zależy od ilości odwiedzin $\boldsymbol{n}$ w stosunku do wszystkich przeprowadzonych rozwinięć $\boldsymbol{N}$. Druga część równania ma formę: $\sqrt{(ln(N)/n)}$. Ta druga część równania odpowiada za pewność na temat wielkości $\boldsymbol{v}$, im więcej próbkowań, tym bardziej jest ona pewna. Całe równanie zaś przedstawia się następująco:

$$UCB1 = v/n + c * \sqrt{(ln(N)/n)} \tag{4.6}$$

Żeby wybrać kolejny węzeł, w którym chcemy przeprowadzić rozwinięcie, musimy znaleźć węzeł z największą wartością UCT, która może być równa np. UCB1. Tu $\boldsymbol{v}$ i $\boldsymbol{n}$ są wartością węzła i ilością rozwinięć z węzła, $\boldsymbol{c}$ jest stałą wymiany pomiędzy pierwszym a drugim wyrażeniem, a $\boldsymbol{N}$ jest liczbą wszystkich przeprowadzonych symulacji w całym drzewie. UCB1 sprawia, że odwiedzane są te węzły posiadające najwyższą wartość oraz te, które były rzadko odwiedzane.

Pozostała nam ostatnia rzecz: jeśli napotykamy na węzeł, którego $\boldsymbol{n}$ jest większe od zera, to znaczy jakieś rozwinięcie było poprzednio wykonane z tego węzła, wtedy rozwijamy węzeł o wszystkie akcje, które możemy wykonać. Np. w przykładzie szachowym tworzymy węzły podrzędne zawierające wszystkie posunięcia konikiem, pionkiem, wieżą itd. Jest to proces dokładnie taki sam jak w poprzednich algorytmach, tylko że dla jednego poziomu. Następnie wybieramy węzeł z największym UCT, w tym wypadku UCT są jednak równe, bo n = 0 więc UCB1 = inf., tak więc pozostaje nam wybrać węzeł spośród węzłów podrzędnych zgodnie z kolejnością alfabetyczną. Z tego węzła przeprowadzamy kolejne rozwinięcie. Ostatecznie po przeprowadzeniu wielu rozwinięć, żeby wybrać ruch, wybieramy spośród węzłów podrzędnych do korzenia, czyli naszego początkowego węzła. Wybieramy ten węzeł z największą wartością rozwinięć $\boldsymbol{n}$, ale tylko dla węzłów bezpośrednio podrzędnych. Poszukiwanie Monte Carlo wraz ze wzrastającą liczbą rozwinięć zbiega się do optymalnego rozwiązania.

5

Rozdział 5

Uczenie ze wzmocnieniem

W przeszłości było wiele prób zbudowania przeróżnych kognitywnych architektur. Miały one tworzyć unifikujący obraz działania mózgu lub oddawać w jakiś sposób jego działanie. Te modele były często luźno inspirowane pewnymi spostrzeżeniami na temat funkcjonowania mózgu i były nastawione na przybliżenie generalnych metod jego działania. Zazwyczaj takie modele wymagały także ogromnych zbiorów stwierdzeń na temat świata w duchu symbolicznego AI. My co prawda w rozdziale o symbolicznym AI skupiliśmy się na może pobocznym temacie, jakim są drzewa poszukiwań, ale w tym dziale często korzysta się ze stwierdzeń zapisanych w formie logiki, tak jak to powiedzieliśmy na początku tamtego rozdziału. Takie symboliczne podejście ma swoje plusy, wykorzystuje, chociażby aparat matematyczny do określania wartości stwierdzeń jest jednak zbyt sztywne dla większości rozwiązań. Sam spróbuj zapisać informacje na jakiś temat za pomocą samych stwierdzeń, żeby przekonać się, że nie jest to łatwe zadanie. Tak więc systemy te wymagały ogromnych zbiorów twierdzeń. Działo się tak, ponieważ te systemy nie posiadały własnego mechanizmu interpretacji tego, co dzieje się w świecie i wyciągania z tego wniosków. Można powiedzieć, że pod pewnymi względami te systemy były nawet w tyle za robotem Shakey, który potrafił interpretować swój prosty świat. Niektóre takie projekty zużyły tysiące godzin pracy wolontariuszy, żeby zwiększyć pojemność baz danych na temat świata, z nadzieją, że jeśli uchwycą ich wystarczającą ilość, to te systemy będą mogły rozumować jak ludzie. Okazuje się, jednak że nikt nie mówi nam nigdy, co się stanie, jeśli będziemy biec ulicą z pełnym wiadrem wody, a jednak wiemy, że nasze nogi będą mokre. Z tego i innych powodów, miliony relacji później ten wysiłek zdaje się bezowocny. Ignorowanie motywacji, uczuć, emocji i stanów umysłu mogło być fundamentalnym powodem porażki tych wysiłków. Jest jednak bardziej podstawowy problem z takim podejściem. Tworzenie dużego systemu z wielu wcześniej zdefiniowanych zasad nie jest proste ze względu na występujące między nimi interakcje. Wyobraź sobie, że próbujesz zbudować samochód od podstaw, nie wiedząc nawet jakie systemy i funkcjonalności powinny zostać dodane. Ta metafora oddaje bezsens próby skonstruowania kopii mózgu od podstaw bez posiadania planu. To podejście nie pozwala na testowanie części przed dodaniem ich do większego systemu i nieprawidłowe działanie jednej z nich oznacza, że całość nadaje się do wyrzucenia, ponieważ nie wiemy która część jest odpowiedzialna za niepowodzenie. Pracując obok

tych wysiłków byli naukowcy tacy jak Richard Sutton, którzy próbowali odkryć podstawowe zasady kierujące zachowaniem inteligentnych systemów. Powzięli oni podejście matematyczne, patrząc się na umysł tylko w celu inspiracji. Te wysiłki okazują się dzisiaj dużo bardziej skuteczne. Niektóre z nich zdają się nawet wyjaśniać niektóre zjawiska występujące w naszej czaszce, jak np. uczenie TD, które zostało połączone z działaniem systemu dopaminergicznego. Dużą przewagą tego podejścia jest również to, że uczenie ze wzmocnieniem, ponieważ używa pewnych pojęć występujących w innych dziedzinach sztucznej inteligencji, może być dość łatwo połączone do innych odkryć takich jak sieci neuronowe. To oferuje efekt synergii i łatwość konceptualną.

5.1 Problem uczenia ze wzmocnieniem

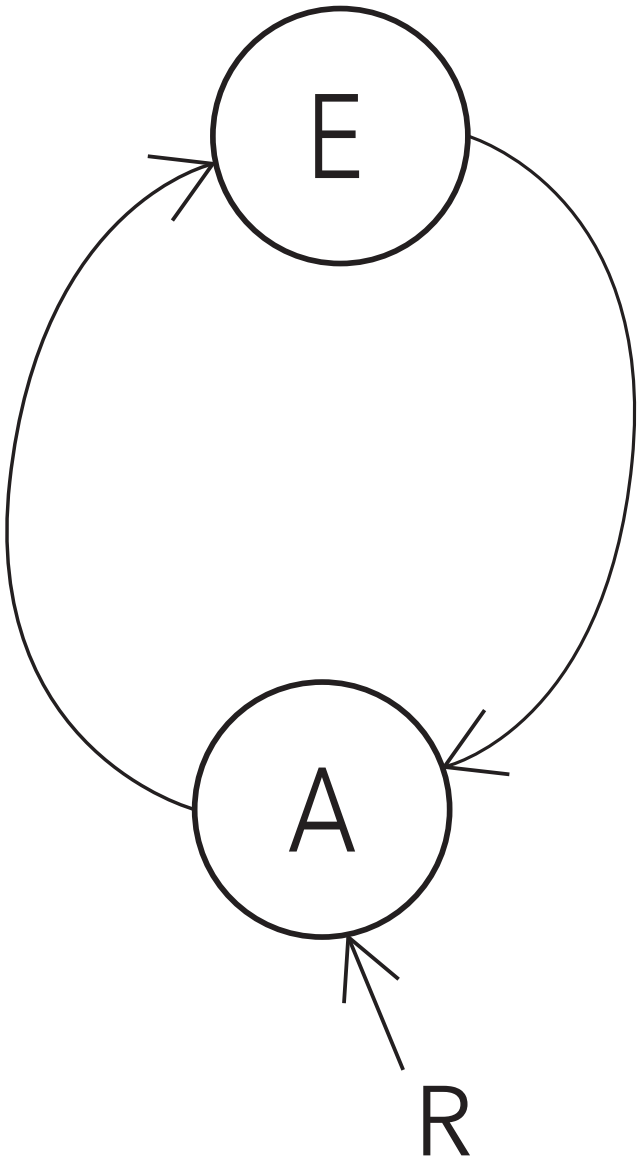
Jak jednak możemy zdefiniować działanie całego inteligentnego systemu, jakim jest nasz mózg za pomocą kilku prostych zasad? Uczenie ze wzmocnieniem (ang. reinforcement learning albo RL w skrócie) identyfikuje kilka niezbędnych składników koniecznych dla inteligentnego zachowania. Na początek potrzebujemy pewnych informacji koniecznych, żeby podjąć decyzję. Ta właściwość jest nazywana **stanem**. Następnie potrzebujemy aktora. Aktor jest osobą, zwierzęciem, robotem, który używa informacji o stanie, żeby podejmować decyzje. Aktor otrzymuje **nagrodę** po każdej zmianie stanu lub po pewnej serii stanów. Nagroda jest podobna do zjawiska przyjemności i bólu, tak więc aktor lubi zwiększać ilość nagrody. Ostatecznie aktor może podejmować pewne **akcje** które wpływają w jakiś sposób na stan. Te akcje mogą być: przesunięciem bierki w szachach, kierunek ruchu ramienia robota lub może wybranie problemu, nad którym powinno się pracować. Ten proces może być zwizualizowany jako pewnego rodzaju okrąg, w którym informacje płyną od środowiska do aktora i do aktora do środowiska. Akcje aktora są modyfikowane przez napływający sygnał nagrody.

Są jeszcze dwa inne kluczowe elementy problemu RL. Tak zwany **zbiór zasad** (ang. policy) jest pierwszą z nich. Zbiór zasad określa dla każdego stanu akcje, które zostaną podjęte przez aktora. Tak więc można myśleć o zbiorze zasad, jak o wyniku działania mózgu.

$$p(s|a) \tag{5.1}$$

Zbiór zasad $\boldsymbol{p}$ dla danego stanu $\boldsymbol{s}$ przy podjęciu akcji $\boldsymbol{a}$.

Ryc. 5.1: Problem uczenia ze wzmocnieniem



Możemy też zidentyfikować funkcję wartości. Poprzednio mówiliśmy o funkcji wartości, ale dla problemu RL możemy określić pewne jej dodatkowe właściwości. Funkcja wartości jest sumą wszystkich sygnałów nagrody od teraz od ostatniego występującego stanu. Możemy to zapisać jako:

$$v(a, a_1, a_2, ...) = \Sigma(r_i, r_{i+1}, r_{i+2}, \dots) \tag{5.2}$$

Gdzie $\boldsymbol{v}$ jest funkcją wartości, $\boldsymbol{a_n}$ są akcjami, które podejmuje aktor, a $\boldsymbol{r_n}$ jest sygnałem nagrody w kroku $\boldsymbol{n}$. Ważne jest, żeby zobaczyć, że chcielibyśmy znać funkcję wartości, ponieważ powiedziałaby nam ona, ile nagrody otrzymamy w przyszłości.

5.2 Wieloręczny bandyta

Wyjaśniliśmy, że całe myślenie aktora może być podsumowane jako zbiór zasad. Jeśli znalibyśmy optymalny zbiór zasad, to moglibyśmy osiągnąć najwyższe możliwe nagrody od teraz do końca wszechświata. Problemem jest jednak to, że większość aktorów wie, na początku, bardzo mało o funkcjonowaniu środowiska, w którym się znajdują, więc nie są w stanie powiedzieć, co jest dla nich najkorzystniejsze. Co powinien zrobić taki aktor, żeby stać się lepszym w poszukiwaniu nagrody? Wyobraź sobie bycie w ogromnym kasynie, gdzie znajduje się wiele automatów do gier zwanych powszechnie jednoręcznymi bandytami (prawdopodobnie dlatego, że każdy, kto wchodzi w bliski kontakt z taką maszyną, zostaje ograbiony). W tym kasynie jest wiele różnych maszyn, każda dająca inną spodziewaną wygraną. My chcemy znaleźć taką, która da nam wygrać lub nie przegrać, jak największą ilość pieniędzy. Może brzmi to tylko jak ładna metafora, ale tak opisany problem nazywany jest problemem **wieloręcznego bandyty** (ang. multi-armed bandit). Zastanówmy się teraz nad najlepszą strategią działania w takim kasynie. Na początku nic nie wiemy o maszynach, także konieczne jest, żebyśmy dokonali pewnej ilości eksploracji, to znaczy, że koniecznym jest aby pociągnąć wajchy należące do wielu jednoręcznych bandytów w poszukiwaniu takiego, który daje dobre nagrody. W ten sposób możemy znaleźć najbardziej opłacalną maszynę, ale jeśli tylko eksplorujemy, to nie otrzymamy korzyści wynikających z wykorzystania takiej maszyny, czyli eksploatacji. Strategia, która zawiera te dwie korzyści, czyli eksplorację i eksploatację nazywana jest **epsilon-chciwa** (ang. epsilon greedy). Określa ona, że powinniśmy eksplorować przez $\boldsymbol{e}$ ilość czasu i eksploatować to, czego się nauczyliśmy przez $\boldsymbol{1 - e}$ ilość czasu. Używając strategii epsilon-chciwej, będziemy otrzymywać wyższe nagrody, wraz z tym, jak będziemy się stawać lepsi w danym zadaniu. Wynika to oczywiście z faktu, że wykonujemy co jakiś czas fazę eksploatacji, w ciągu której wykorzystujemy najlepszy automat. Jak ustalić przez ile czasu powinno się to jednak odbywać? Zazwyczaj ustawilibyśmy $\boldsymbol{e}$ tak, żeby równała się pewnej małej wartości takiej, jak $\boldsymbol{e}$=0.1. To oznacza, że eksplorujemy przez tylko 10% czasu. W długim okresie wystarczy to jednak do znalezienia optymalnego rozwiązania. Jednym problemem tego podejścia

jest to, że nawet jeśli znamy spodziewaną wartość każdej z maszyn z wielką dokładnością, to nadal będziemy zmuszeni poświęcać 10% czasu na eksplorowanie, co będzie nas powstrzymywać przed otrzymaniem najwyższej możliwej nagrody. Tu pojawia się modyfikacja strategii epsilon-chciwej. W niej będziemy zmniejszać $\boldsymbol{e}$ po pewnej ilości $\boldsymbol{N}$ prób. Za każdym razem, gdy wykonaliśmy wielokrotność $\boldsymbol{N}$ prób zmniejszymy epsilon, używając równania:

$$e = y * e \tag{5.3}$$

Gdzie $\boldsymbol{y}$ jest pewną wartością pomiędzy 0 a 1.

To pozwoli nam wykonywać mniej i mniej eksploracji, w miarę tego, jak stajemy się lepsi w wykonywaniu zadania, jednocześnie zwiększając naszą nagrodę albo jak w przypadku jednoręcznych bandytów – zyski. Jest jeszcze jedna rzecz, którą moglibyśmy zrobić. Jeśli znamy najwyższą możliwą wartość funkcji wartości, to możemy zmniejszać parametr $\boldsymbol{e}$ proporcjonalnie do poprawy w funkcji wartości. Jest to pewnego rodzaju strategia równoważenia ilości eksploracji z wielkością funkcji wartości. To pozwoli nam przeznaczać na eksplorację ilość czasu równą procentowi nieosiągniętej maksymalnej wartości. Jeśli funkcja wartości osiągnęła np. 30%, to eksplorujemy przez 70% czasu. Kiedy poprawimy się, w tym co robimy, to zmniejszymy ilość eksploracji, bo będzie ona mniej korzystna, jako że częściowo wiemy już jak osiągać pewien procent wartości. Przed wprowadzeniem jeszcze jednego sposobu powtórzmy to, czego już się nauczyliśmy, a mianowicie, że funkcja wartości jest zdefiniowana jako:

$$v(a, a_1, a_2, ...) = \Sigma(r_i, r_{i+1}, r_{i+2}, \dots) \tag{5.4}$$

Podane równanie, żeby otrzymać funkcję wartości, sumuje wszystkie nagrody od teraz do nieskończoności z równą wagą. Jednak w prawdziwym życiu troszczymy się dużo bardziej o nagrodę w momencie $\boldsymbol{r_i}$ niż tą w momencie $\boldsymbol{r_{i+1000}}$ Dlaczego tak się dzieje? Jednym z powodów jest zapewne fakt, że przyszłość nie jest tak pewna, jak teraźniejszość, więc z chęcią zamienimy nagrodę daleko na horyzoncie na mniej oddaloną, bo ta odległa może nigdy nie nadejść. Dlatego powinniśmy dodać pewien parament określający pewność co do przyszłości. Nazwijmy go $\boldsymbol{d}$ tak jak we współczynniku dyskontowym. Pomnóżmy $\boldsymbol{r_{i+1}}$ przez $\boldsymbol{d}$, $\boldsymbol{r_{i+2}}$ przez $\boldsymbol{d * d}$, $\boldsymbol{r_{i+3}}$ przez $\boldsymbol{d * d * d}$ itd.

$$v(a, a_1, a_2, ...) = \Sigma(r_i, r_{i+1} * d, r_{i+2} * d^2, \dots) \tag{5.5}$$

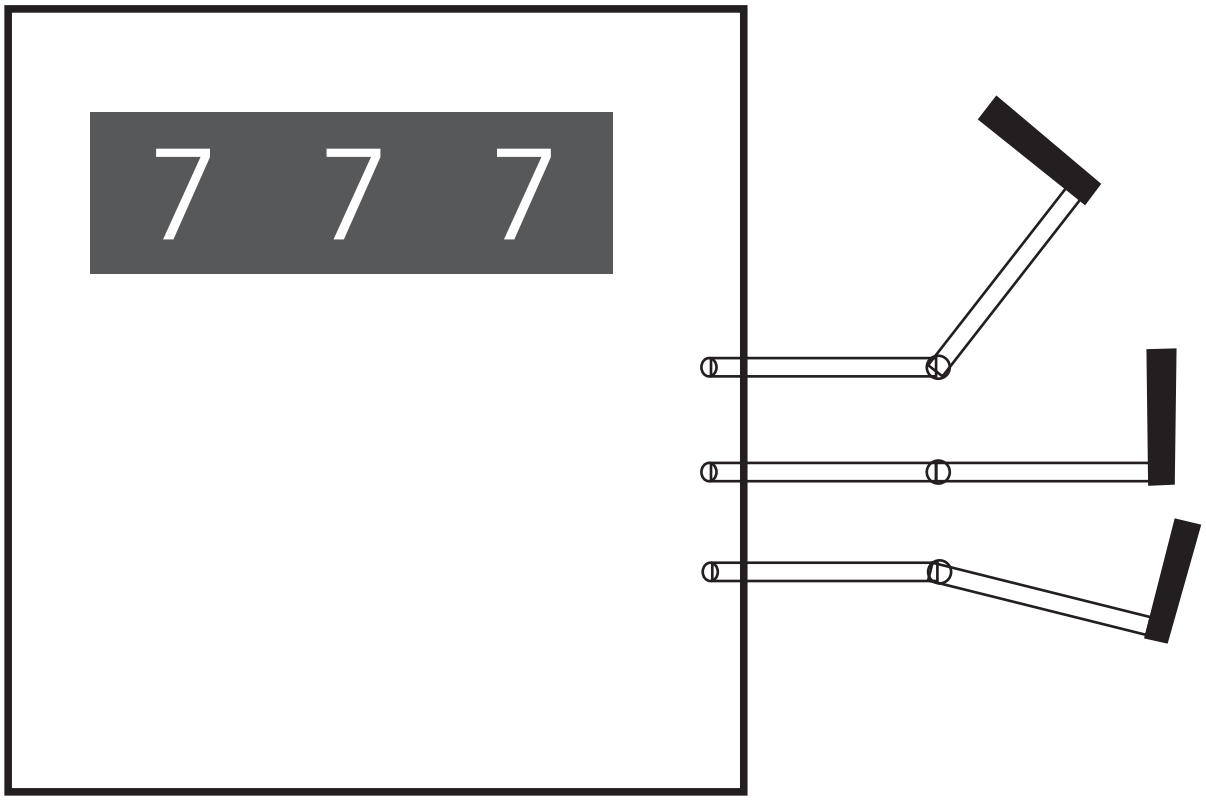
To jest bardziej realistyczny pogląd na naszą funkcję wartości. Teraz pamiętając o tym, możemy stworzyć sieć neuronową, która będzie przybliżać tę funkcję wartości. Mając tę sieć, będziemy eksploatować wieloręcznego bandytę, używając sieci neuronowej do aproksymacji jakości każdego jednoręcznego bandyty. To podejście jednak nie zadziała, ponieważ nie daliśmy naszej sieci żadnych przykładów do trenowania. Dlatego nadal

potrzebujemy eksplorować, żeby uzyskać przykłady, na których będziemy trenować. Wytrenujemy taką sieć, podając przykłady sytuacji, które napotkaliśmy i odpowiadającą im sumę nagród jak pokazano wyżej. Żeby wybrać pomiędzy stanem eksploracji i eksploatacji, użyjemy formuły UCT takiej jak w podrozdziale 4.6 dotyczącym MCTS. Dla każdego "bandyty" obliczymy i wybierzemy tego z najwyższą wartością:

$$UCB1 = v + c * \sqrt{ln(N)/n} \tag{5.6}$$

Gdzie $\boldsymbol{v}$ będzie rezultatem otrzymanym z naszej sieci, $\boldsymbol{c}$ stałą eksploracji, $\boldsymbol{N}$ całkowitą liczbą prób, a $\boldsymbol{n}$ ilością prób wykonanych na konkretnym automacie.

Ryc. 5.2: Wieloręczny bandyta



5.3 MDP

Łańcuch Markowa jest modelem środowiska, w którym następny stan zależy tylko od obecnego stanu i od prawdopodobieństwa zmiany tego stanu. Z tego powodu jest nazywany procesem stochastycznym. Wyobraźmy sobie, dla przykładu, przewidywanie pogody. Nasz model może pokazywać, że jeśli mamy ładną pogodę to występuje 7% szansy na to, że jutro będzie padać. Jeśli jednak już dzisiaj pada, to występuje 13% szansy, że pojawi się burza, 36% szansy, że jutro też będzie padać i 51% szans, że jutro się rozpogodzi. Taki model jest właśnie nazywany łańcuchem Markowa. Może on się jednak składać z wielu więcej stanów i przejść między nimi. Możemy zaprezentować taki model graficznie jako graf z węzłami stanu połączonymi krawędziami związanymi z prawdopodobieństwem.

$$(S, P) \tag{5.7}$$

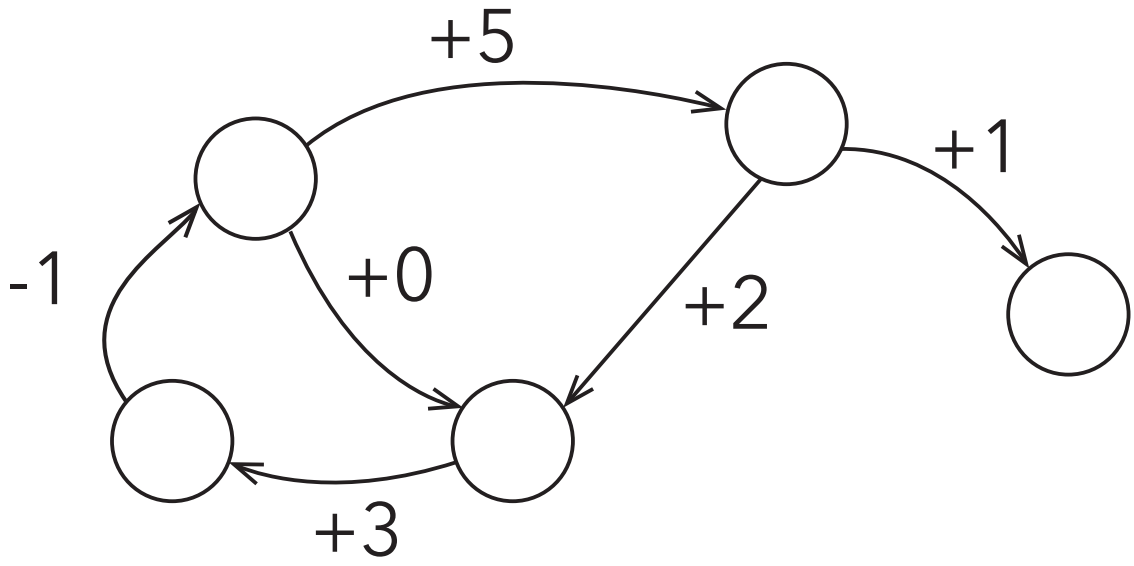
$\boldsymbol{S}$ jest stanem, $\boldsymbol{P}$ jest prawdopodobieństwem zmiany stanu.

Problem decyzyjny Markowa (ang. Markov decision process, MDP dla krótkości zapisu) jest rozszerzeniem pomysłu łańcucha Markowa, które dodaje akcje i nagrody. Ten model jest wyjątkowo podatny na idee uczenia ze wzmocnieniem. W MDP mamy stany, które prowadzą z pewnym prawdopodobieństwem do akcji, które znów prowadzą z innymi prawdopodobieństwami do pewnych stanów. Każda zmiana stanu związana jest z nagrodą. Rozszerzmy nas przykład dot. pogody o jej przewidywanie. Powiedzmy, że jeśli jest ładna pogoda i poprawnie przewidzimy, że jutro też będzie ładna, to otrzymujemy nagrodę +1. Jeśli jednak przewidzimy, że będzie ładnie, a w rzeczywistości będzie padać, to otrzymujemy nagrodę -2. Następnie przechodzimy do kolejnego stanu, w zależności od tego, co rzeczywiście się stało i tak, jeśli trafimy np. na deszcz, to otrzymujemy nagrodę -2, przesuwamy się do węzła z deszczem. Teraz mamy 3 przewidywania: ładna pogoda, deszcz, burza. Jeśli przewidzimy ładną pogodę, to otrzymujemy +2, jeśli to samo zrobimy z deszczem to +3, jeśli przewidzimy burzę +5. Jednak jeśli nie uda się nam przewidzieć sytuacji, to otrzymamy -1 za źle przewidzianą ładną pogodę, -2 za deszcz i -5 za burzę. Następnie przechodzimy do kolejnego węzła, według tego, co naprawdę się wydarzyło. Jeśli zdarzyła się akurat burza, to kończymy, bo od burzy, w naszym przykładzie, nie prowadzą żadne strzałki. Oznacza to, że jest ona węzłem końcowym.

$$(S, A, P, R) \tag{5.8}$$

$\boldsymbol{S}$ jest stanem, $\boldsymbol{P}$ jest prawdopodobieństwem zmiany stanu, tak jak w łańcuchu Markowa, $\boldsymbol{A}$ są akcjami, które aktor może wybrać, a $\boldsymbol{R}$ są nagrodami związanymi z przeszłym i obecnym stanem.

Ryc. 5.3: Przykładowe przejścia między stanami MDP



5.4 Uczenie Monte-Carlo

Jedną z metod uczenia zbioru zasad oraz funkcji wartości jest **uczenie Monte-Carlo**. Działa ono poprzez patrzenie się jaką kumulatywną nagrodę następującą po stanie $\boldsymbol{s}$ otrzymał aktor. Jest to niedokładne przybliżenie funkcji wartości. Jak pamiętamy, suma wszystkich nagród jest równa funkcji wartości. Mogłoby się nam wydawać, że suma nagród będzie dokładnie odzwierciedlać funkcję wartości i możemy uczyć na tych pojedynczych przykładach z powodzeniem. Tak jednak nie jest, ponieważ po nastąpieniu wydarzenia, które śledzimy, następują też inne wydarzenia, które mają wpływ na wynik sumy nagród. Dlatego bierzemy średnią z wielu takich wydarzeń, aby otrzymać przybliżony wynik funkcji wartości. Jeśli weźmiemy wiele przykładów to wydarzenia następujące po naszym wydarzeniu, którym jesteśmy zainteresowani, się uśrednią. To sprawi, że przewidywanie będzie bliższe rzeczywistej oczekiwanej wartości wydarzenia. Dodawanie kolejnych przykładów do naszego przewidywania na temat jednego wydarzenia daje nam lepsze i lepsze prognozy na temat rzeczywistej funkcji wartości. Jest jednak pewien problem z tym podejściem. Chodzi tu o sposób sumowania nagród. Jak możemy zsumować niekończące się nagrody? Jednym z problemów uczenia Monte-Carlo jest to, że historia potrzebuje mieć punkt końcowy. Dzieje się tak, ponieważ nie możemy sumować nieskończonej liczby nagród bez ucinania reszty w jakimś arbitralnym punkcie, żeby być w stanie nauczyć się czegokolwiek.

$$v(s) = AVG(v_1, v_2, v_3, \dots) \tag{5.9}$$

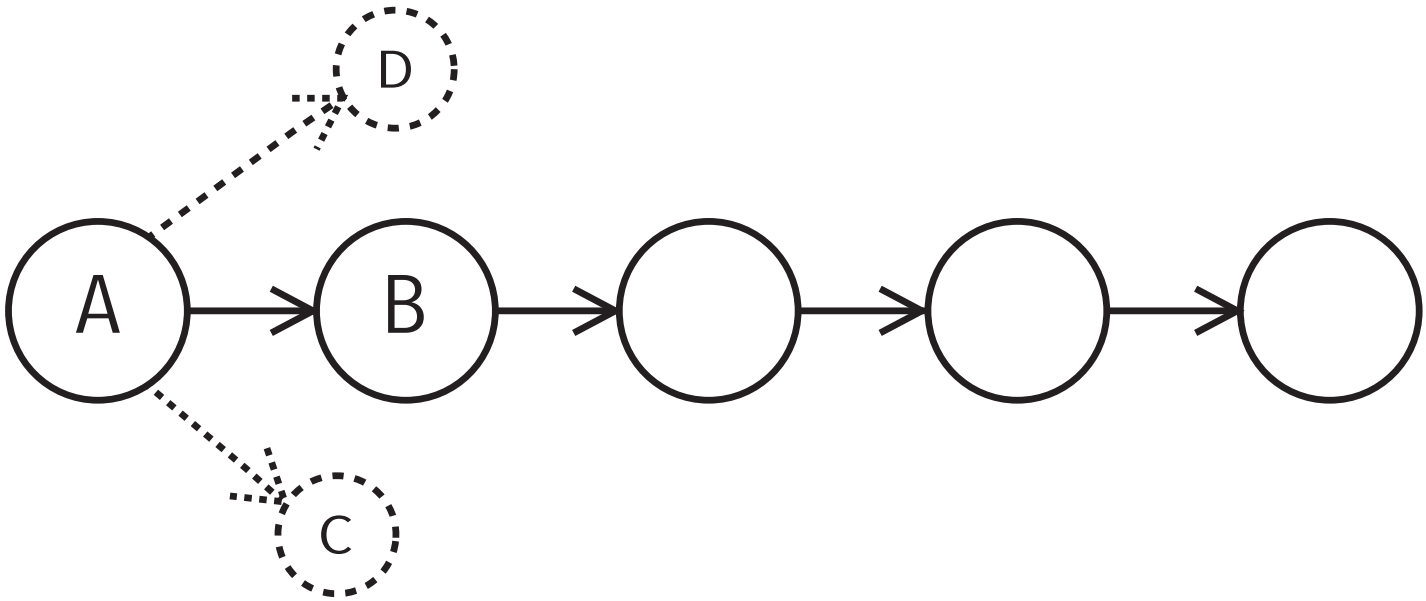
Uczenie Monte-Carlo, gdzie $\boldsymbol{v_1}$, $\boldsymbol{v_2}$ itd. są sumami nagród w konkretnym epizodzie, a $\boldsymbol{v}$ jest funkcją wartości otrzymaną przez wzięcie średniej z tych sum.

Powiedzmy, jak moglibyśmy zaimplementować uczenie Monte-Carlo. Możemy sobie wyobrazić posiadanie wielkiej tablicy z osobnym rekordem dla każdego stanu $\boldsymbol{s}$. W każdej komórce umieszczalibyśmy tą średnią. Teraz, aby aktualizować wartość każdej z tych komórek, potrzebujemy zasady aktualizacji. Moglibyśmy ją otrzymać, utrzymując w pamięci dwie informacje: jedną z sumą wszystkich nagród w różnych epizodach i drugą z ilością epizodów, które sprawdziliśmy. Dzieląc pierwszą wartość przez drugą, otrzymalibyśmy wynik. Jest jednak inna metoda dająca ten sam wynik:

$$u_{avg} = \frac{x + (k-1) * u_{oldAvg}}{k} \tag{5.10}$$

Gdzie $\boldsymbol{u_{avg}}$ jest nową średnią, $\boldsymbol{u_{oldAvg}}$ bieżącą średnią, $\boldsymbol{k}$ jest liczbą epizodów, a $\boldsymbol{x}$ jest nowym przykładem sumy nagród, który napotkaliśmy.

Ryc. 5.4: Rozwinięcie Monte-Carlo



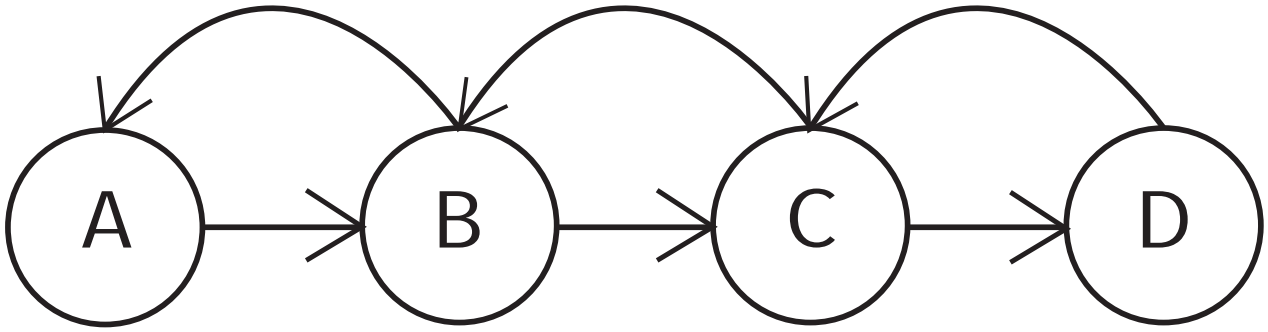
To podejście jest możliwe dla małych problemów, jednak kiedy problem rośnie do dużych rozmiarów, jak na przykład podczas gry w szachy, staje się niemożliwym, aby utrzymywać rekord każdego możliwego stanu. Jak moglibyśmy przybliżyć działanie takiej tablicy? Tu na pomoc przychodzi generalna **aproksymacja funkcji**. W naszym przypadku użyjemy jednego z aproksymatorów jakim jest sieć neuronowa. Sieci neuronowe są tak nazywane, ponieważ w teorii mogą reprezentować każdą możliwą funkcję z pewną dokładnością. Tak więc będziemy używać sieci neuronowej do aproksymowania działania tabeli przeglądowej. Jakie będzie wejście, wyjście i nagroda dla naszej sieci? Wejściem będzie stan. Wyjście określimy jako wartość funkcji wartości dla danego stany wejściowego. Tak np. wejściem może być pozycja szachowa, a wyjściem numeryczna ocena pozycji przez arcymistrza szachowego. Żeby przybliżać wynik otrzymywany przez propagację przez sieć, do oceny profesjonalisty, użyjemy funkcji wartości, która będzie używać różnicy między wyjściem a sumą nagród dla danego epizodu do swojego uczenia. Następnie odejmiemy ocenę profesjonalisty od oceny sieci i wszystko to podniesiemy następnie do kwadratu. Taka funkcja nazywana jest **średnim błędem kwadratowym** (ang. mean squared error lub MSE).

$$MSE = (u_{expected} - u_{net})^2 \tag{5.11}$$

$\boldsymbol{U_{expected}}$ jest sumą nagród albo funkcją wartości dla naszego epizodu, $\boldsymbol{u_{net}}$ jest wyjściem naszej sieci neuronowej. Musimy tylko pamiętać, że jeśli nasz epizod jest nieskończony, będziemy musieli w pewnym momencie uciąć resztę nagród.

Taka funkcja wartości zapewni nam dobre uczenie ze względu na to, że kara, jaką będzie otrzymywać sieć, będzie zależeć od odległości między oceną sieci a oceną spodziewaną. Zamiast oceny profesjonalisty możemy wziąć wynik gry, co jest bardziej realne, ponieważ zazwyczaj nie mamy zbioru pozycji ocenionych przez profesjonalistę. Jeśli tak właśnie jest, to możemy wziąć jak największą liczbę rozegranych gier z ich wynikami. Wybrać spośród nich pewną liczbę pozycji z wynikami i na takiej podstawie uczyć naszą sieć. Zauważmy, że popularne pozycje takie jak np. 1.e4 2.e5 mogą się powtórzyć w naszym zbiorze wielokrotnie. Na dodatek mogą mieć one różne wyniki, ponieważ niektórzy gracze mogli wygrać a inni przegrać z tej samej pozycji. Nie przeszkadza to jednak w uczeniu, ponieważ sieć tak jak wcześniej tablica uśredni te wyniki.

Ryc. 5.5: Propagacja błędu w TD



5.5 Uczenie TD

Alternatywą dla metody Monte-Carlo jest metoda **temporalnych różnic** (ang. temporal difference, TD dla krótkości zapisu). W metodzie TD szukamy różnic pomiędzy funkcją wartości dla następujących po sobie stanów. Wyobraźmy sobie na przykład, że gramy w szachy. W tym rodzaju gry nagroda pojawia się tylko na końcu i nie ma pośrednich nagród, za powiedzmy zbicie pionka. Potrzebujemy ich jednak, żeby dokonywać wyboru. To ważny problem, na który zwracano uwagę dużo wcześniej. Chociażby Newell miał wątpliwości czy w wyniku wygrana, remis, przegrana jest wystarczająca ilość informacji, żeby czegokolwiek być w stanie się nauczyć. Żeby rozwiązać ten problem, moglibyśmy w pewnym sensie propagować wstecznie nagrody od końca epizodu do jego początku. Jeśli to zrobimy, to każdy stan poprzedzający koniec gry będzie mieć wartość równą wynikowi gry. Pozycji powiedzmy 1.e4 2.e5, przypiszemy wartość równą temu, co było wynikiem całej gry. Teraz moglibyśmy użyć tablicy, żeby zapisywać rezultaty dla kolejnych sprawdzanych pozycji, ale już mówiliśmy, przy okazji omawiania metody Monte-Carlo, że istnieje lepszy sposób. Użyjemy do przewidywania sieci neuronowej. Wejście i wyjście będą takie same jak poprzednio. Jedynym co ulegnie zmianie, będzie błąd.

$$MSE = (u_{net} - u_{net-1})^2 \tag{5.12}$$

To jest funkcja błędu dla uczenia TD. $\boldsymbol{u_{net}}$ jest wartością funkcji wartości dla obecnego epizodu a $\boldsymbol{u_{net-1}}$ jest podobnie wartością funkcji wartości dla poprzedniego epizodu.

Zobaczmy, jak działa to uczenie. Bierzemy obecne przewidywanie co do wyniku, które zwraca nam sieć i przewidywanie tej sieci ruch wcześniej. Obliczamy różnicę za pomocą równania (5.12). Następnie używamy wyniku do uczenia sieci na temat obecnej pozycji. Widzimy więc, że sieć próbuje przewidywać wynik samej siebie. Możesz teraz powiedzieć, że całe to uczenie nie ma sensu, bo jak nasza sieć ma się nauczyć od samej siebie mimo tego, że nic na początku nie wie. I masz rację, na początku uczenie nie będzie dobre, ale będzie występowało, ponieważ na końcu epizodu zamiast $\boldsymbol{u_{net}}$ użyjemy realnego rezultatu. Jak już jednak powiedzieliśmy, uczenie to nie będzie z początku najwyższej jakości. Ten problem nazywany jest z ang. biase'em uczenia TD. Mocną stroną tego podejścia wobec metody Monte-Carlo jest jednak to, że nie musimy czekać do końca epizodu, aby nauczyć się czegoś, uczenie odbywa się po każdej zmianie stanu. To jest niezwykle korzystne w sytuacjach, gdy nagroda jest rzadka tak jak w przykładzie z szachami. Kluczowym jednak pytaniem jest to czy istnieje różnica, między tym, czego nauczy się Monte-Carlo a TD? Zobaczmy przykład:

Mamy dwa wydarzenia A i B oraz połączone z nimi odpowiednie nagrody. Dane czterech epizodów wyglądają następująco:

A 0, B 0 end

B 1 end

B 1 end

B 0 end

Teraz chcemy zobaczyć jaką odpowiedź dostaniemy od MC i od TD. MC (skrót od metody Monte-Carlo) wyciągnąłby średnią z wszystkich przykładów, dając nam:

A = 0

B = 1/2

Metoda TD dałaby nam inną odpowiedź:

A = 1/2

B = 1/2

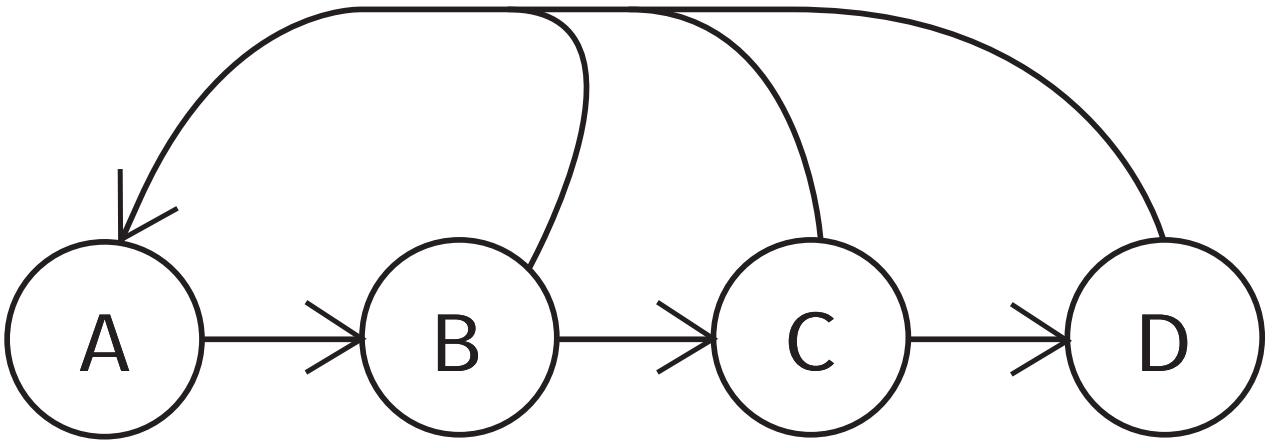
Więc jak TD dało nam tę odpowiedź? Jeśli pamiętasz, jak mówiliśmy o MDP w rozdziale 5.3, to możesz sobie wyobrazić działanie metody TD jako wybieranie najbardziej prawdopodobnego MDP związanego z problemem. W naszym przypadku MDP wyglądałby następująco:

A przechodzi w B 100% razy.

B daje nam 1 z 50% prawdopodobieństwem.

Widzimy, więc że rozwiązania obu tych metod nie są takie same. Mówimy, że MC zbiega się do rozwiązania, które ma najmniejszy błąd kwadratowy a metody TD do najprawdopodobniejszego modelu MDP.

Ryc. 5.6: Propagacja błędu w TD(λ)



5.6 Uczenie TD(lambda)

Poprzednio widzieliśmy dwa różne podejścia do problemu uczenia ze wzmocnieniem, ale którego powinniśmy używać? Odpowiedź brzmi: nie musimy wybierać, ponieważ istnieje metoda łącząca te dwa sposoby. Kiedy używaliśmy metody Monte-Carlo, patrzyliśmy się na każdą nagrodę do końca epizodu. Kiedy natomiast używaliśmy metody TD, patrzyliśmy się tylko jeden krok do przodu. Co, jednak jeśli moglibyśmy się popatrzeć gdzieś pomiędzy? Nie do końca epizodu, ale więcej niż jeden krok. To podejście nazywane jest **n-krokową** metodą TD. Żeby otrzymać naszą funkcję nagrody, do której będziemy się starali dopasować działanie aproksymatora, dodamy **n** kroków nagród tak jak w MC, a na końcu zamiast reszty epizodu dodamy funkcję wartości w kroku **n+1** tak jak w uczeniu TD. Teraz, jeśli wybierzemy **n** = 0 to metoda ta jest ekwiwalentna do metody TD, a jeśli wybierzemy **n = inf.** to jest taka sama jak MC. Wiemy eksperymentalnie, że zazwyczaj najlepsze rozwiązania leżą gdzieś pomiędzy tymi dwoma ekstremami. Jak jednak mamy wybrać konkretną liczbę kroków, której będziemy używać? Tu znów okazuje się, że niekoniecznie musimy wybierać. A to ponieważ algorytm **TD(lambda)** łączy wszystkie wartości **n** kroków w jedną wartość docelową. Działający tu mechanizm jest bardzo prosty. Weźmiemy wszystkie możliwe wartości dla **n**-kroków, zaczynając na **n** = 0 a kończąc na **n** = inf. Dodamy je wszystkie, otrzymując nową wartość. Aby jednak nie brać pod uwagę odległych wydarzeń, w powiedzmy 100-kroku, z taką samą wagą jak np. w 4-kroku wprowadzono współczynnik lambda. Lambda zawiera się w przedziale 0 do 1. Określa ona, jak bardzo powinniśmy brać pod uwagę odległe konsekwencje. Z jednej strony przecież chcemy je brać pod uwagę, ale z drugie im bardziej odległe, tym mniej pewne jest, co dokładnie je spowodowało. Musimy więc ustalić, jak bardzo zależy nam na konkretnej właściwości. Zrobimy więc odrobinę inaczej. Weźmiemy wszystkie możliwe wartości dla n-kroków i dodamy je pomnożone przez $\boldsymbol{\lambda^n}$. Ten współczynnik będzie zmniejszać wagę dużych, a więc odległych $\boldsymbol{n}$-kroków. Na końcu wynik przez $\boldsymbol{(1-\lambda)}$ dla obliczeniowej konsystencji.

$$G = (1 - \lambda) * \sum_{i=0}^{n}((\sum_{j=0}^{i}(r_j) + v(i)) * \lambda^i) \tag{5.13}$$

Gdzie $\boldsymbol{\Sigma(r_j)}$ oznacza sumę nagród w $\boldsymbol{j}$ krokach, $\boldsymbol{v(i)}$ funkcja wartości w **i**-tym kroku, a $\boldsymbol{G}$ jest wartością docelową, którą chcemy przybliżać, używamy jej więc w równaniu MSE.

Możemy teraz użyć G w równaniu na MSE:

$$MSE = (G - u_{net})^2 \tag{5.14}$$

$\boldsymbol{U_{net}}$ jest tu oczywiście wynikiem działania naszej sieci.

Ten algorytm został wykorzystany w słynnym programie TD-Gammon stworzonym w 1992 roku przez Geralda Tesauro. Nazwa programu pochodziła właśnie od uczenia TD oraz od nazwy gry, w której ten program konkurował, a mianowicie Tryktraka (ang. backgammon). Gra ta w wielkim uproszczeniu polega na podróży wokół planszy w kierunku przeciwnym do konkurenta swoimi pionami. Celem gry jest zdjęcie wszystkich swoich pionów z planszy. Aby to uczynić, obaj gracze rzucają dwoma kośćmi i poruszają się o wyrzuconą liczbę pól. W czasie gry piony wchodzą ze sobą w interakcje. Co ciekawe można wygrać normalnie lub poprzez zdobycie tzw. gammonu. Ze względu na losowość występującą w grze, która powoduje szybkie rozgałęzianie, a więc powstanie wielu możliwych historii, głębokość drzewa poszukiwań jest z konieczności ograniczona. W programie TD-Gammon była ona ograniczona do ruchów i odpowiedzi przeciwnika. Następnie pozycje były oceniane przez sieć neuronową. Sieć neuronowa programu była nauczona za pomocą algorytmu TD(λ). Program nie wykorzystywał wiedzy zdobytej przez ludzi, a uczył się bezpośrednio ze swoich gier. W wyniku powstania programu TD-Gammon zmieniła się teoria grania w tryktroka. Gra pozycyjna programu jest oceniana przez niektórych ekspertów jako wyśmienita, przewyższająca nawet ich własną.

Rozdział 6

Gry i więcej

Pokryliśmy w tym momencie szeroki zakres tematów, zaczynając na sieciach neuronowych i optymalizowaniu ich działania, co daje nam moc znajdywania relacji między strumieniami danych. Patrzyliśmy się na drzewa poszukiwać, które utrzymują w pamięci rozwiązania, których próbowaliśmy i pomagają nam eksplorować przestrzeń możliwości. Widzieliśmy także, w jaki sposób uczenie ze wzmocnieniem daje nam unifikujący pogląd na aktorów uczących się w rzeczywistych środowiskach i zobaczyliśmy problemy, które w nich napotykają. Wspomnieliśmy też czasami o historycznych wydarzeniach ważnych dla dziedziny sztucznej inteligencji. W tym rozdziale przeglądniemy kilka prac badawczych od firmy Deepmind, mając na uwadze to, czego nauczyliśmy się do tej pory. Po przeczytaniu tego rozdziału powinieneś być zapoznany, przynajmniej na wysokim poziomie z najnowszymi badaniami w tej dziedzinie. Pospekulujemy również na końcu tego rozdziału nad przyszłymi możliwymi podejściami do tak zwanej generalnej sztucznej inteligencji tj. **AGI**. Ta część tekstu jest może najważniejsza ze wszystkich. Budując na tym, co zostało powiedziane wcześniej, daje nietechnicznemu czytelnikowi wgląd w to, co dzieje się pod maską sztucznej inteligencji, uczenia maszynowego oraz głębokich sieci neuronowych. Odbiegając od tematu, zwrot głęboka sieć neuronowa (ang. deep neural network) znaczy tylko tyle że sieć ma więcej niż jedną warstwę neuronów. To tyle. Jednak ktoś niezaznajomiony z tematem mógłby być pod wrażeniem, kiedy usłyszałby o głębokich sieciach neuronowych. Mam nadzieję, że teraz rozumiesz głębiej, jak wygląda rzeczywistość i nie będziesz tak łatwo manipulowany przez medialne próby łapania twojej uwagi. Mamy także nadzieję, że niektóre obawy związane ze sztuczną inteligencją zostały przez nas rozwiane i możesz krytykować rozwiązania AI na wyższym poziomie zrozumienia. Mamy nadzieję, że będziesz w stanie wyjaśnić kilka rzeczy swoim znajomym. Ostatecznie mamy nadzieję, że będziesz w stanie podejmować lepsze decyzje, używając tej wiedzy. Czy to będzie poprzez dalsze pogłębianie wiedzy i tworzenie nowych rozwiązań, przygotowanie lepszego planu dla swojej firmy, inwestowanie w przedsięwzięcia związane z technologią, czy nawet decyzje dotyczące życia osobistego takie jak wybór platform społecznościowych, z których chcesz korzystać, ze względu na wykorzystywanie w nich tych metod. Jesteśmy głęboko przekonani, że większa wiedza może cię prowadzić do lepszego życia dla ciebie i innych. Popatrzmy teraz na obecny stan badań.

6.1 AlphaGo (rok 2016)

AlphaGo jest algorytmem, który został stworzony z myślą o pokonaniu gry planszowej, która nie poddawała się technikom sztucznej inteligencji przez najdłuższy czas, mowa tu o Go. W tamtym czasie naukowcy myśleli, że znaczący przełom w dziedzinie jest konieczny, aby osiągnąć ten cel oraz że jest on odległy o przynajmniej dekadę. W tym czasie brytyjska firma Deepmind przejęta przez Google ciężko pracowała, aby udowodnić coś przeciwnego. Sprawdzili oni najlepsze w tamtym czasie podejścia do tej gry, które używały MCTS wzmocnionego przez sieć zbioru zasad, która miała przewidywać ruchy, które zagrałby człowiek. Badzacze zaproponowali ich własny sposób, który miał używać sieci zbioru zasad i funkcji wartości połączonych z MCTS, a to wszystko wytrenowane w części przez uczenie ze wzmocnieniem. Jak pokazują w pracy na temat AlphaGo, gra Go ma bardzo dużą przestrzeń możliwości, w której trzeba wyszukiwać rozwiązanie. Jej szerokość $\boldsymbol{b}$ (ilość legalnych ruchów dla danej pozycji) wynosi około 250, a głębokość $\boldsymbol{d}$ (długość gry), która wynosi średnio 150. To jest dużo większe drzewo możliwości niż np. szachy, które mają $\boldsymbol{b}$ równe mniej więcej 35, a $\boldsymbol{d}$ równe około 80. Z tego właśnie powodu proste wyszukiwanie alfa-beta nie jest wystarczające, aby rozwiązać problem gry Go. Nie możemy popatrzeć się wystarczająco głęboko i nie mamy prostej funkcji wartości, tak jak w szachach gdzie każdej figurze możemy przypisać pewną wartość. W Go wartość każdego ruchu zależy głównie od innych ruchów, które zostały wcześniej wykonane. Żeby nauczyć AlphaGo, Deepmind wytrenował kilka sieci neuronowych. Na początku zebrali zbiór danych zawierający gry profesjonalistów. Następnie wytrenowali coś, co nazwali siecią 'SL policy' żeby przewidywała ruchy zrobione przez tych graczy. Ta sieć miała 13 warstw i trenowała na 30 milionach przykładów. Naukowcy wytrenowali dodatkowo mniejszą i szybszą sieć do wykonywania tego samego zadania. 'SL policy' udało się otrzymać wynik 57% poprawnie przewidzianych ruchów, a mniejszej sieci udało się to w 24.2% przypadków. Kolejno wytrenowano 'SL policy' sposobem uczenia przez wzmacnianie. Pozwolono sieci wybierać ruchy i na końcu gry, gdy znana jest nagroda, dokonywano uczenia ze wzmocnieniem. Użyta metoda opierała się o podobne mechanizmy jak te, na które patrzyliśmy tylko dla sieci zbioru zasad zamiast dla funkcji wartości. Wytrenowana w ten sposób sieć RL była w stanie pokonać sieć SL w 80% przypadków. Ludzie pracujący nad projektem chcieli jednak aby użyć także funkcji wartości. Żeby otrzymać zbiór danych, pozwolili grać sieci RL wiele gier ze samą sobą aż do momentu ich zakończenia. Następnie wybierali niektóre pozycje spośród tego zbioru danych, żeby uniknąć nadmiernego dopasowania ze względu na korelacje pomiędzy pozycjami wewnątrz jednej gry. Funkcją błędu była MSE względem wyniku gry, który był równy +1 dla wygranej i -1 dla przegranej (w grze Go nie ma remisów). Architektura sieci symulującej funkcję wartości była podobna do sieci 'SL policy'. Oczywista różnica, jaka między nimi występowała to fakt, że 'SL policy' zwracała dystrybucję prawdopodobieństwa a funkcja

wartości jedną liczbę. Sieć 'policy' była przydatna w zmniejszaniu szerokości drzewa, a funkcja wartości w zmniejszaniu głębokości drzewa poszukiwań. W pracy na temat AlphaGo badacze zwracają uwagę, że funkcja wartości miała podobną siłę do sieci RL używającej rozwinięć Monte-Carlo, ale używała 15 000 razy mniej mocy obliczeniowej. Ostatecznie połączyli oni sieci neuronowe z drzewem poszukiwań, a mianowicie algorytmem MCTS wcześniej opisanym w rozdziale 4.6. Zamiast używać formuły UCB, wybierają oni węzeł za pomocą następującego równania:

$$a_t = argmax_a(Q(s,a), +u(s,a)) \tag{6.1}$$

Gdzie $\boldsymbol{s}$ jest stanem, $\boldsymbol{a}$ akcją, a $\boldsymbol{argmax_a}$ oznacza wybór $\boldsymbol{a}$ z największą wartością $\boldsymbol{Q + u}$ które są opisane poniżej. Wyrażenie $\boldsymbol{Q(s,a)}$ jest zdefiniowane jako:

$$Q(s,a) = 1/N(s,a) * \Sigma(1(s,a,i) * V(s)) \tag{6.2}$$

Gdzie $\boldsymbol{1(s,a,i)}$ jest funkcją charakterystyczną zbioru, która określa czy węzeł był odwiedzony podczas **i-tej** symulacji. $\boldsymbol{V(s)}$ jest funkcją wartości, a $\boldsymbol{N(s,a)}$ jest liczbą wizyt w danym węźle.

Drugie wyrażenie w równaniu (6.1) jest zdefiniowane:

$$u(s,a) = c * P(s,a) * \sqrt{\sum^{b} N(s,b)/N(s,a)} \tag{6.3}$$

Gdzie $\boldsymbol{c}$ jest stałą eksploracji, $\boldsymbol{P(s,a)}$ jest wcześniejszym prawdopodobieństwem, które otrzymaliśmy za pomocą policy użytej na węźle nadrzędnym, i $\boldsymbol{N(s,b)}$ jest liczba wizyt do możliwych ruchów z naszego węzła, a $\boldsymbol{N(s,a)}$ jest liczbą wizyt w naszym węźle.

Wartość $\boldsymbol{V(s)}$ z równania (6.2) jest kombinacją wartości wyjścia z sieci $\boldsymbol{v_\theta}$ oraz wartości rozwinięcia (ang. rollout) $\boldsymbol{z}$ które są połączone w następujący sposób:

$$V(s) = (1 - \lambda) * v_\theta + \lambda * z \tag{6.4}$$

Gdzie $\boldsymbol{\lambda}$ jest parametrem mieszania.

W równaniu (6.1) podobnie jak w MCTS $\boldsymbol{u}$ jest częścią równania odpowiadającą za eksplorację, natomiast $\boldsymbol{Q}$ jest częścią równania określającą wartość węzła. $\boldsymbol{Q}$ jest po prostu wartością $\boldsymbol{V}$ znormalizowaną przez ilość rozwinięć, tak żeby ich ilość nie wpływała na wynik. Jeśli chodzi o $\boldsymbol{u}$ (6.3) to, zamiast określać nasz brak wiedzy na temat węzła tylko przy pomocy ilości rozwinięć, używana do określenia tej potencjalnej wartości jest również wartość otrzymana z sieci zbioru zasad. To ona szybko, za jednym razem, określa wartość wszystkich ruchów na planszy. W równaniu (6.4) widzimy, że zamiast wykorzystywać samą wartość rozwinięcia, jak to miało miejsce w klasycznym

MCTS, wykorzystywana jest również informacja na temat pozycji pozyskana z sieci funkcji wartości.

Po fazie rozwinięcia wszystkie parametry i niektóre ukryte parametry muszą być przesłane do nadrzędnych węzłów. Nie będziemy tu wchodzić w szczegóły. Jedyną rzeczą, którą musimy teraz dodać, jest odpowiedź na pytanie, kiedy rozwijać węzeł. Ekspansja jest wykonywana, kiedy wartość wizyt w węzle $\boldsymbol{N(s,a)}$ jest większa niż pewna wartość $\boldsymbol{n_{threshold}}$. Jako najlepszy węzeł jest wybierany węzeł z największą ilością wizyt. Pokrótce to jest całość algorytmu AlphaGo. Został on użyty w Marcu 2016, żeby pokonać byłego mistrza świata Lee Sedola w serii pięciu meczów. Jedyna gra wygrana przez człowieka, gra czwarta, została wygrana dzięki specjalnej taktyce Sedola, która miała na celu skomplikowanie gry. Podczas tego wysiłku znalazł on ruch nazywany „boskim ruchem". AlphaGo oceniała go jako bardzo mało prawdopodobny. Prawdopodobnie z powodu tego błędu komputer zagrał poniżej oczekiwań w kolejnych kilku posunięciach, tracąc w efekcie wiele punktów na rzecz człowieka i w konsekwencji przegrywając całą grę. Później zespół AlphaGo znalazł i poprawił błąd, który powodował dziwne zachowanie.

6.2 AlphaZero (rok 2017)

W 2017 roku zespół Deepmind wziął się za rozwiązywanie szachów i gry Shogi używając podobnych technik do tych użytych w AlphaGo, tym razem jednak miano nie wykorzystywać wiedzy ludzi do nauki, a zamiast tego programy miały się nauczyć grać od podstaw na ponadludzkim poziomie. Normalne silniki szachowe używają techniki wyszukiwania alfa-beta wraz z bardzo dokładnie ręcznie dopasowaną funkcją wartości, książką otwarć i końcówek oraz korzystają z wielu heurystyk, które poprawiają działanie systemu w wielu małych okolicznościach. AlphaZero przyniosła zupełnie nowy sposób myślenia o problemie, pozwalając na stworzenie silnika szachowego Lc0, który jest w stanie konkurować z najlepszym klasycznym silnikiem szachowym, mianowicie Stockfishem. Leela Zero była w stanie osiągnąć to w bardzo krótkim czasie rozwoju i w przyszłości będzie prawdopodobnie w stanie przekonywająco pokonać Stockfisha. **AlphaZero** korzysta z architektury pod wieloma względami podobnej do programu AlphaGo. Używa sieci 'policy', sieci wartości oraz algorytmu MCTS. Największą różnicą pomiędzy nimi jest metoda trenowania obu modeli. Kiedy AlphaGo było trenowane przy użyciu gier pochodzących od ludzi, AlphaZero uczy się tabula rasa, znając na początku tylko zasady gry. Wyszukiwanie jest wg autorów zaimplementowane podobnie jak w AlphaGo. Żeby stać się lepszym, AlphaZero grało przeciwko sobie i było wzmacniane -1 dla przegranej, 0 dla remisu i +1 dla wygranej. Parametry sieci były uczone przy użyciu MSE + Cross entropy (inny typ funkcji błędu) + regularyzacja L^2, która została opisana w rozdziale 3.6. Najbardziej interesującym pomysłem jest

to, że parametry w sieci zbioru zasad są aktualizowane tak, aby zbliżyć je do prawdopodobieństwa po użyciu wyszukiwania, więc w pewnym sensie sieć zbioru zasad stara się przewidzieć co stanie się podczas wyszukiwania. Dzięki temu przy wykorzystaniu, sieć zbioru zasad jest w stanie zredukować sprawdzane przypadki do tylko do tych najbardziej obiecujących. AlphaZero wyszukuje 80 tys. pozycji na sekundę w porównaniu do 70 milionów, które sprawdza Stockfish. Jednak mimo tego AlphaZero była w stanie pokonać Stockfisha w meczu 1000 gier. Rezultat wyniósł 155 zwycięstw dla AlphaZero, 6 dla Stockfisha, reszta to remisy.

6.3 MuZero (rok 2019)

Architektura **MuZero** jest pod pewnymi względami podobna do AlphaZero, ale w niektórych znacząco się różni. MuZero używa podobnej wersji MCTS. Łączy też 'policy' i funkcję wartości w jedną sieć z dwoma wyjściami.

$$(6.3.1)p, v = f(s) \tag{6.5}$$

Gdzie $\boldsymbol{p}$ jest policy, $\boldsymbol{v}$ funkcją wartości, $\boldsymbol{f}$ siecią neuronową, a $\boldsymbol{s}$ jej wejściem. Największą zmianą są nowe funkcje dynamiki i reprezentacji. Funkcja dynamiki jest używana do przewidywania nagrody w następnym kroku. Stan jest ustawiony na równy funkcji reprezentacji.

$$(6.3.2)s_0 = h(o\_1, o\_2, ..., o\_t) \tag{6.6}$$

Zamiast używać danych jako wejścia, używana jest funkcja reprezentacji $\boldsymbol{h}$ która bierze jako wejście dane wejściowe $\boldsymbol{o\_t}$ z poprzednich $\boldsymbol{t}$ kroków.

To jest najbardziej ekscytująca idea w MuZero. Oznacza ona właściwie, że możemy wykonywać wyszukiwanie w sytuacjach, w których było to poprzednio niemożliwe, takich jak np. gry Atari. W grach Atari nie posiadamy zasad pozwalających nam na generowanie następnej pozycji z pozycji obecnej i wybranego ruchu. To jest miejsce, gdzie funkcja reprezentacji jest wykorzystywana. Raczej niż polegać na zasadach do generowania przyszłych pozycji, wykorzystujemy do tego celu funkcję reprezentacji. To pozwala na wyszukiwanie w sytuacjach, w których generowanie drzewa poszukiwań było poprzednio niemożliwym. Żeby docenić to podejście, pomyślmy o sytuacji, w której gramy np. w grę komputerową. Nie znamy nigdy dokładnych zasad, na których opiera się ta gra. Jedyne co wiemy na początku to kilka komend kontrolnych, które podał nam twórca gry. Reszty musimy się domyślić, opierając się na naszej wiedzy o prawdziwym świecie, a może także o innych grach, które napotkaliśmy wcześniej. Tę funkcję spełnia funkcja reprezentacji. Mając taką funkcję reprezentacji, możemy przeprowadzać planowanie za pomocą drzewa poszukiwań w sutuacji, w której poprzednio nie było

to możliwe. Możemy np. wyobrazić sobie, że jeśli strzelimy do strażnika, to inni strażnicy zostaną poinformowani o czyhającym na nich zagrożeniu. Tak więc wybieramy inną ścieżkę dostępną na drzewie poszukiwań, które zostało wygenerowanie dzięki funkcji reprezentacji. To podejście osiągnęło nowy rekord w rozwiązywaniu gier Atari, jak jest to wyjaśnione w pracy na temat MuZero. Jedynym pytaniem pozostaje jak wytrenować taki model. Jak możemy przeczytać, wszystkie parametry funkcji reprezentacji, dynamiki i predykcji są trenowane razem. Błąd z wszystkich sieci jest sumowany i dodawana jest regularyzacja L^2. MuZero używa drzewa poszukiwań MCTS, używając funkcji predykcji, reprezentacji i dynamiki. Następnie po $\boldsymbol{n} = 800$ symulacji, ruch jest wybierany. Ta procedura jest wykonywana przez **K** kroków gry. Następnie, żeby uczenie było możliwe, najbliższa gałąź MCTS jest porównywana do prawdziwej gry. To porównanie daje nam dane treningowe dla funkcji błędu. To porównywanie gałęzi jest kolejną ekscytującą ideą w architekturze MuZero. Sprawia to, że może się ono uczyć nie tylko na bezpośrednio wykonanym ruchu, ale na całej sekwencji zasymulowanych ruchów. Porównując to, co zostało zasymulowane, z tym, co stało się naprawdę, uzyskujemy błąd. Ten błąd z każdego z przykładów wykorzystujemy do trenowania sieci. To podejście sprawia, że możemy uzyskać dużo więcej przykładów testowych. MuZero wydaje się niezwykle ciekawe, ponieważ zdaje się, że może być użyte nie tylko do gier ze znanymi zasadami, ale także do każdego innego problemu uczenia ze wzmocnieniem. MuZero wygląda jak spojrzenie na to, jak systemy AI będą wyglądać w przyszłości.

6.4 AGI

Generalna sztuczna inteligencja (ang. artificial general intelligence, AGI) jest ideą, którą mówi, że jest możliwym stworzyć program, który będzie w stanie rozwiązać dowolny problem, który mu zaprezentujemy. Część opisanego postępu zdaje się poruszać w stronę większej ogólności jak na przykład AlphaZero, które można wytrenować, aby grało w kilka różnych gier. Jest jednak również prawdą, że wielu naukowców ma duże wątpliwości na temat możliwości osiągnięcia AGI w bliskiej przyszłości. AI posunęło się na wielu frontach w czasie ostatnich kilku dekad. Poprawiliśmy metody optymalizacji, żeby być w stanie lepiej odpowiadać na intuicyjne pytania. Poprawiliśmy algorytmy wyszukiwania tak, aby mogły lepiej pracować z intuicjami sieci neuronowych. Zbudowaliśmy także zaawansowaną formalną strukturę, dzięki której możemy myśleć o problemie uczenia ze wzmocnieniem. Nadal jest jedna duża rzecz, która zdaje się nam uciekać. Jest nią język. Każdy człowiek na Ziemi komunikuje się, używając pewnego rodzaju języka, i mamy trudności z wyobrażeniem sobie jak wyglądałoby nasze życie bez niego. To jest według autora dosyć silny argument za koniecznością języka w naszych systemach. Czym w ogóle jest **język**? Język jest zbiorem znaków połączonych z pewnymi znaczeniami. Możemy używać tych znaków, żeby opisywać

sytuacje, przewidywać kolejno następujące znaki i do generowania odpowiedzi na pytania skonstruowane za pomocą tych znaków. Jak jednak wcześniej widzieliśmy, idee o wielkiej mocy muszą być dodane do potężnych struktur, żeby były szeroko aplikowalne. Zobaczmy, jak te symboliczne pomysły mogłyby być użyte w połączeniu z sieciami neuronowymi, żeby dać najlepsze rezultaty: Czy pamiętasz, jak silnik szachowy wyszukiwał miliony przykładów na sekundę, ale rozwiązania oparte o sieci neuronowe wykonywały dużo mniej takich wyszukiwań, osiągając mimo tego dobre rezultaty ze względu na użycie sieci neuronowych do oceny sytuacji? Możemy myśleć o drzewach poszukiwań jako, w pewnym sensie, bardzo podobnych do języka. Ostatecznie przecież w wielu modelach języka zdanie może być zapisane jako swego rodzaju drzewo. Jeśli weźmiemy ten pomysł krok dalej, to możemy stwierdzić, że te pomysły są dokładnie takie same. To znaczyłoby, że ludzie wykonują nawet mniej wyszukiwania w porównaniu do MuZero, jednocześnie przewyższając je w większości dziedzin. Znaczyłoby to także że język jest rodzajem algorytmu poszukiwań kierującym nas w wyborze następnego węzła do rozwinięcia. Jednak język ma pewne właściwości, których naszym systemom nadal brakuje. Jak możemy opisać język jako algorytm, tak jak robiliśmy to wcześniej dla różnych metod? Na początku stwierdzilibyśmy autorytatywnie, że słowa mogą być reprezentowane jako pewne punkty w $\boldsymbol{n}$ wymiarowej przestrzeni. To może się nie wydawać intuicyjne, ponieważ słowa wydają się dyskretnymi bytami, ale może to wynikać z używania tylko pewnych części dostępnej przestrzeni. Następną rzeczą, którą byśmy wprowadzili, jest to, że każdy węzeł powinien posiadać korespondujące do niego jedno słowo, które go opisuje. Oczywiście możemy dać listę słów albo opis sytuacji za pomocą całego zdania, albo paragrafu, ale może wyraża to tylko pierwsze słowo z większą dokładnością. Słowa są też połączone ze sobą. Wszystkie takie relacje mogłyby teoretycznie być wytłumaczone jako podrzędność, nadrzędność, równoważność węzłów. Tak więc słowo byłoby określone jako lista zawierająca dwie wartości: znaczenie $\boldsymbol{m}$ i jego relację $\boldsymbol{r}$ do jakiegoś innego słowa.

$$(6.4.1) w = (m, r) \tag{6.7}$$

Jak więc ta reprezentacja byłaby używana? Wyróżnilibyśmy cztery główne sieci. Sieć funkcji wartości, która korzystając z wewnętrznej reprezentacji 'X', zwracałaby spodziewaną wartość sytuacji opisanej przez wewnętrzną reprezentację. Ta sieć działałaby tak jak każda poprzednio opisana sieć wartości, tylko wykorzystywałaby wewnętrzną reprezentację jako wejście. Następnie sieć 'P' która brałaby wewnętrzną reprezentację i zwracała następne słowo, które najlepiej opisuje daną sytuację. Ta sieć byłaby siecią tworzącą kolejne słowa. Wykorzystywałaby reprezentację wewnętrzną do określania jakie słowo powinno nastąpić na następnym miejscu. Słowo składa się ze znaczenia, które jest punktem w $\boldsymbol{n}$ wymiarowej przestrzeni oraz z relacji, która określa miejsce na grafie, które powinno być wyszukiwane w następnej kolejności. Tak więc sieć ta wybierałaby najpierw punkt w **n**-dim a następnie inne, wcześniej istniejące słowo, dla którego

nowe słowo będzie w relacji podrzędnej. Trzecia sieć 'R' podobna do funkcji reprezentacji z MuZero brałaby jako wejście listę wszystkich słów wypowiedzianych przez 'P' oraz odpowiadające im wartości określone przez sieć funkcji wartości i zwracałaby wewnętrzną reprezentację. Te trzy sieci byłyby trenowane łącznie. Dodatkowym wejściem do sieci funkcji wartości byłoby znaczenie najnowszego wypowiedzianego słowa, czyli punkt w **n**-dim. To sprawiałoby, że funkcja wartości wiedziałaby, z jakiego rodzaju problemem ma do czynienia. Czwartą ostatnią siecią byłaby sieć zbioru zasad. Ponieważ wskazywanie kolejnych relacji $\boldsymbol{r}$ odbywałoby się na zasadzie wskazania węzła nadrzędnego, od którego chcemy przeprowadzić rozwinięcie (co byłoby osiągnięte jako dystrybucja prawdopodobieństwa nad słowami), to musielibyśmy wybrać spośród kilku węzłów podrzędnych do rozwinięcia. W tym zadaniu pomagałaby właśnie sieć zbioru zasad. Mogłaby być ona trenowana łącznie z innymi sieciami lub też osobno. Użycie wszystkich tych sieci, jeśli wszystko zadziałałoby zgodnie z zamierzeniem, doprowadziłoby do stworzenia swego rodzaju języka, którym nauczyłby się operować nasz system.

To, co zostało opisane w tym rozdziale (6.4), jest czymś przypominającym Science-Fiction. Są w nim oczywiście pewne naukowe pomysły, ale wzięte w przyszłość, po to, aby stworzyć przekonywającą historię. Autorzy nie znają żadnej implementacji pomysłów zawartych w tym rozdziale i obecnie pozostają one tylko spekulacją.

Podziękowania

Na końcu chciałbym podziękować osobom bez których ta książka nie wyglądałaby tak jak to co możecie czytać. Po pierwsze dziękuję panu Michałowi Dyzmie za przygotowanie pliku LaTeX, dzięki czemu powstała cała część techniczna, a więc odpowiednie formatowanie i wygląd tekstu. Dziękuję mu też za rady, które pomogły mi w stworzeniu tej książki. Następnie chciałbym również podziękować pani Ninie Szul za wykonanie ilustracji oraz okładki. Mam nadzieję że dzięki nim książka jest jaśniejsza oraz ładniejsza.