Uniwersytet Jagielloński

Wydział Matematyki i Informatyki

Zespół Katedr i Zakładów Informatyki Matematycznej

Sieć neuronowa dla Renju

Szymon Stankiewicz

Nr indeksu: 1101451

Promotor
Dr Piotr Micek

Opracowano zgodnie z Ustawą o prawie autorskim i prawach pokrewnych z dnia 4 lutego 1994 r. (Dz.U.1994 nr 24 poz. 83) wraz z nowelizacji z dnia 25 lipca 2003r. (Dz.U.2003 nr 166 poz. 1610) oraz z dnia 1 kwietnia 2004 r. (Dz.U.2004 nr 91poz. 869)

Kraków, 2017

Streszczenie

Sieci neuronowe znalazły szerokie zastosowanie w wielu gałęziach informatyki - począwszy od analizy obrazów, poprzez przetwarzanie języka naturalnego, a skończywszy na sterowaniu robotami.
Jednym z głośniejszych zastosowań tej rodziny modeli w ostatnich latach był projekt *AlphaGo*, którego efektem było stworzenie programu zdolnego do pokonania profesjonalnego gracza w starochińskiej grze *Go*, która przez wiele lat stanowiła, kolejne po szachach, wyzwanie dla sztucznej inteligencji.
Zainspirowany powyższym sukcesem przedstawię w poniższej pracy model zgadujący optymalne ruchy dla gry planszowej *Renju*, którego skuteczność wynosi **73,34%**.

Spis treści

1 Wstęp

1.1 Cel i inspiracje

Uczenie głębokie odnalazło zastosowanie w szerokim wachlarzu problemów. Jednym z nich jest tworzenie sztucznej inteligencji do gier, która jest w stanie konkurować z profesjonalnymi graczami. Zainspirowany sukcesem systemu AlphaGo [11] obrałem sobie za cel przeniesienie pierwszego etapu tej publikacji, polegający na zgadywaniu ruchów profesjonalistów na podstawie bazy rozgrywek, na grunt innej gry planszowej - Renju.

1.2 Renju

Renju jest japońską grą planszową dla dwóch graczy, rozgrywaną na planszy zwanej *gobanem*, mającej 225 pól tworzących kwadrat o wymiarach 15 na 15. Gracze na zmianę kładą na wolnych polach planszy pionki (zwane kamieniami) w swoim kolorze. Celem gry jest utworzenie ze swoich kamieni nieprzerwanej linii o długości przynajmniej 5.
Jeśli ograniczylibyśmy zasady tylko do tych podanych powyżej otrzymalibyśmy zasady innej gry - *Gomoku*. *Renju*, będące profesjonalnym wariantem *Gomoku*, definiuje dodatkowe zasady, których celem jest wyrównanie rozgrywki. Ponieważ rozpoczynający, grający czarnymi kamieniami, gracz posiada strategię wygrywającą dla podstawowego wariantu gry [2] takie dodatkowe zasady są koniecznością na profesjonalnym poziomie. Zasady *Renju* nakładają na czarnego gracza zakazy wykonania następujących ruchów:

- *double three* - czyli ułożenia dwóch, krzyżujących się w tym miejscu, nieprzerwanych przez białe kamienie linii składających się z 3 czarnych kamieni,
- *double four* - podobnie jak przy *double three* tylko dla czterech kamieni,
- *overline* - czyli ułożenia 6 lub więcej kamieni w jednej, nieprzerwanej linii.

Dodatkowo biały gracz wygrywa gdy:

- ułoży nieprzerwaną linię z sześciu lub więcej kamieni,
- zmusi czarnego gracza do wykonania zabronionego ruchu.

Zależnie od wariantu dodawane są kolejne zasady, które opisują pierwsze ruchy. Przykładem takiego zestawu dodatkowych zasad jest *RIF opening rule*:

1. Tymczasowy czarny gracz kładzie trzy pierwsze kamienie zgodnie z jednym z 26 podstawowych otwarć.
2. Tymczasowy biały gracz wybiera czy dalej pozostaje biały czy zamienia się z tymczasowym czarnym graczem.
3. Biały gracz kładzie czwarty kamień.
4. Czarny gracz podaje dwie propozycje piątego ruchu.
5. Biały gracz wybiera jedną z nich i wykonuje szósty ruch.
6. Od tej pory gra rozgrywa się zgodnie z podstawowymi zasadami *Renju*.

1.3 Sieci neuronowe

Sieci neuronowe są modelami, których zadaniem jest odnajdywanie wzorców w danych wejściowych. Dzięki temu możliwe jest predykowanie etykiet, którymi te dane są oznaczone. Na rodzinę sieci neuronowych składa się wiele znacznie różniących się od siebie modeli takich jak:

- sieci gęste,
- sieci rekurencyjne [7],
- sieci konwolucyjne [10].

Wspólnym mianownikiem tych modeli jest przede wszystkim sposób nauki. Opierając się na funkcji kosztu, która definiuje jak dobrze "radzi" sobie model (im większa wartość tym gorzej) możemy znaleźć gradient dzięki któremu aktualizujemy iteracyjnie wewnętrzne wagi w modelu. Algorytm ten nazywa się *gradient descent*.

2 Dane

Dane na których pracowałem pochodzą ze strony `http://www.renju.net/`. Jest to baza partii rozegranych przez profesjonalnych graczy na turniejach Renju. Ponieważ turnieje rozgrywane są na różnych zasadach początkowych pominąłem te turnieje, które posiadały specjalne zasady dla więcej niż 6 pierwszych ruchów. W pozostałych turniejach pominąłem pierwsze 6 ruchów, aby nie zaburzać procesu uczenia. Podział danych na zbiory treningowe, testowe i walidacyjne wyglądał następująco:

Zbiór	Liczba gier	Liczba ruchów	%
trenigowy	38 441	1 145 608	79,86%
walidacyjny	4 805	145 200	10,12%
testowy	4 806	143 771	10,02%
razem	4 052	1 434 579	100%

Ponieważ obrót planszy o dowolny kąt będący wielokrotnością 90° i lustrzane odbicie generują również poprawne stany gry, możliwa była augmentacja danych, dzięki czemu powiększyłem zbiór treningowy 8-krotnie (4 obroty i 2 odbicia).

3 Features

Każde pole planszy, przed zaaplikowaniem modelu, zostaje zakodowane jako wektor 5 wartości.

#	Nazwa	Wartości	Opis
1	Ja	0/1	1 jeśli na tym polu jest mój kamień
2	Przeciwnik	0/1	1 jeśli na tym polu jest kamień przeciwnika
3	Puste	0/1	1 jeśli to pole jest puste
4	Czarny	0/1	1 jeśli czarny gracz wykonuje ruch
5	Jedynki	1	1 zawsze

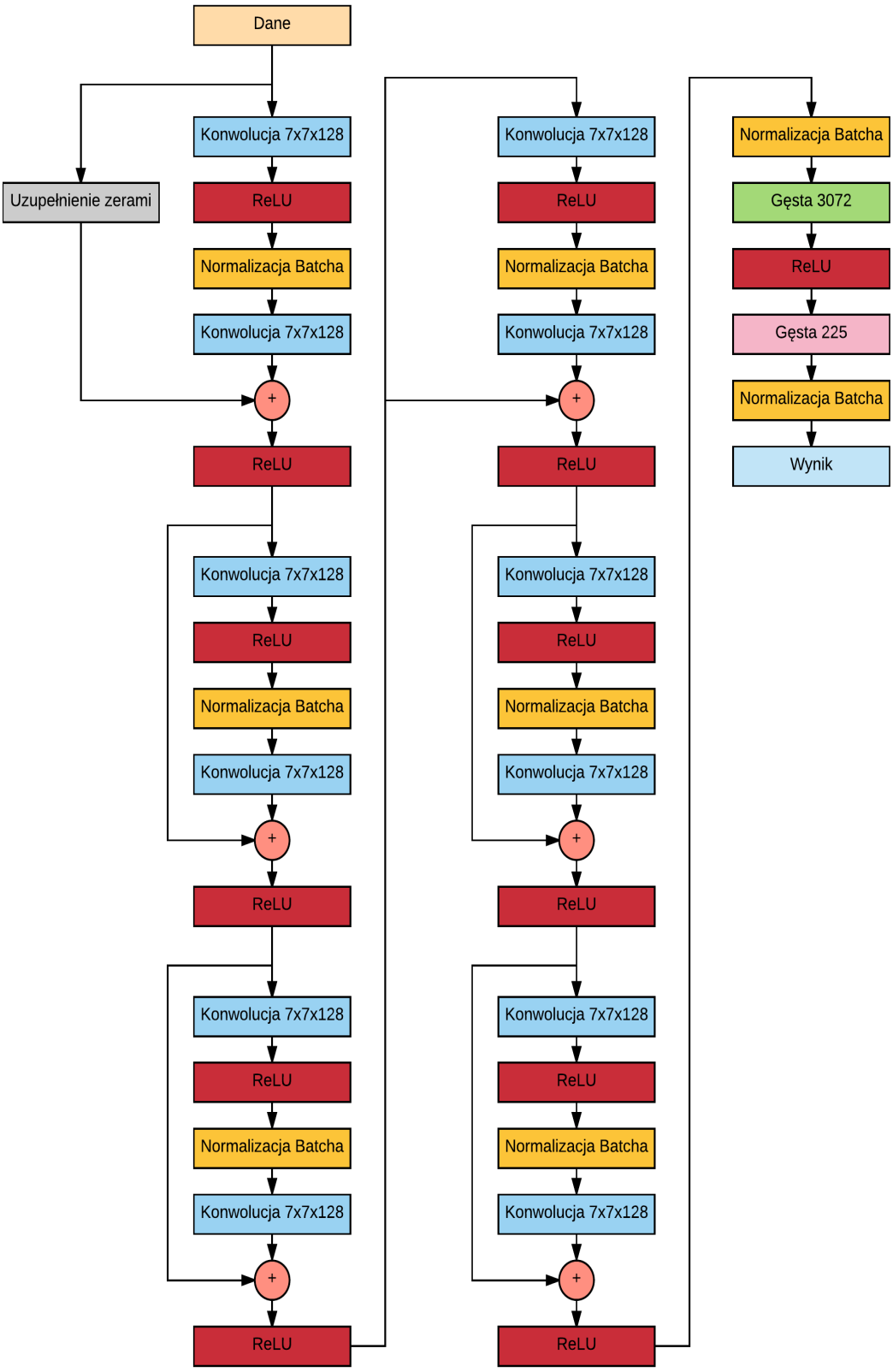
Taka reprezentacja jest standardowym zabiegiem w sieciach neuronowych - umożliwia ona sieci łatwiejsze dobieranie wag dzięki rozbiciu danych na wektory. Zadanie pierwszych czterech wartości jest dość oczywiste, natomiast zastanawiać może ostatnia wartość - najprawdopodobniej służy ona do rozróżniania pól z wnętrza i zewnętrza planszy, gdy przyłożony filtr wyjdzie poza planszę. Pojawienie się warstwy wypełnionej jedynkami zainspirowana jest doborem feature'ów w AlphaGo.

4 Architektura Modelu

Model wykorzystany w tej pracy jest konwolucyjną siecią neuronową z rezydualnymi połączeniami [5]. Pierwszym krokiem modelu jest zamiana wejściowych danych o wymiarach 15x15x5 na dane o rozmiarze 15x15x128 poprzez dodanie zer na nowych warstwach. Następnie dane przepuszczone są sześciokrotnie przez rezydualne połączenia. Na każdą rezydualną warstwę składa się przyłożenie dwuwymiarowej konwolucji rozmiaru 7 na 7 ze 128 filtrami, aktywacja ReLU, normalizacja batcha [8] i ponowne przyłożenie dwuwymiarowej konwolucji rozmiaru 7 na 7 ze 128 filtrami. Między kolejnymi połączeniami nie stosuję warstwy poolingowej. Po skończeniu warstw rezyzudalnych dane są spłaszczane i przepuszczane przez dwie warstwy gęste o rozmiarach odpowiednio 3072 i 225. Sieć była uczona przy użyciu algorytmu Adam [9] z początkowym *learning rate* ustawionym na 0.001.
Przy ostatecznej ewaluacji model uruchamiany jest na wszystkich możliwych obrotach i odbiciach planszy. Dodawany do tego jest również wynik dla planszy z zamienionymi pozycjami czarnych i białych kamieni. Ostatecznym wynikiem jest suma predykcji na tych danych - wybierany jest ten ruch, którego wartość jest największa.

4.1 Warstwy Konwolucyjne

Warstwy konwolucyjne są rozwiązaniem szeroko wykorzystywanym w analizie obrazów. Polega ono na "przykładaniu" tego samego filtru w różne miejsca obrazu. Przyłożenie filtru $\mathcal{F}$ (będącego



Rysunek 1: Architektura modelu

3-wymiarowym tensorem) na danej pozycji polega na wzięciu tensora z danych wejściowych o rozmiarach odpowiadających filtrowi (jeśli $\mathcal{F}$ ma wymiary $n \times m \times k$ i chcemy przyłożyć go do tensora $\mathcal{T}$ na pozycji i, j to mówimy tutaj o podtensorze $\mathcal{T}[i : i + n; j : j + m; 0 : k]$) i policzeniu iloczynu Frobeniusa (sumy iloczynów wartości na odpowiadających pozycjach) tego tensora i filtru $\mathcal{F}$.
Intuicyjnie warstwy konwolucyjne uczą się znajdować proste i lokalne wzorce w danych wejściowych - składając wiele takich warstw jesteśmy w stanie odnajdywać coraz bardziej zaawansowane wzorce nie zwiększając znacząco liczby wartości, których sieć musi się wyuczyć.
Warstwy konwolucyjne często korzystają w swoich definicjach z jeszcze dwóch wartości. Jedną z nich jest *padding* - jego wartość definiuje jak bardzo filtr może "wystawać" poza dane wejściowe. Można zauważyć, że jeśli nie pozwolimy filtrowi wystawać to wyjściowy tensor zmniejszy swój rozmiar, aby temu zapobiec dodajemy do wejściowego tensora pewną wartość na brzegach pierwszych dwóch wymiarów (najczęściej są to zera), co pozwoli nam sterować rozmiarem wyniku. W moim modelu *padding* ustawiony jest tak aby nie zmieniać wymiarów (czyli na rozmiar filtra minus 1). Drugą jest *stride*, którego zadaniem jest powiedzenie jak daleko od siebie chcemy przykładać do siebie filtry. W moim modelu *stride* ustawiony jest na 1, tak aby przykładać filtr w każdym możliwym miejscu.

4.2 ReLU

ReLU jest funkcją aktywacji zdefiniowaną w nastepujący sposób:

$$ReLU(x) = max(0, x)$$

definiuje się również funkcję *dziurawego* ReLU w następujący sposób:

$$ReLU_\alpha(x) = max(\alpha x, x)$$

gdzie α jest pewną małą, dodatnią liczbą rzeczywistą. Zastosowanie drugiego wariantu pozwala nam na pozbycie się problemu z zerowym gradientem dla x mniejszego od 0, ja jednak w moim modelu skorzystałem ze standardowego ReLU.
Funkcje takie jak ta pozwalają nam na dodanie nieliniowości do naszej sieci. ReLU jest bardzo często wykorzystywaną funkcją w sieciach neuronowych - była ona odpowiedzią na problem znikającego gradientu [6] przy korzystaniu z takich funkcji jak sigmoid.
Wykorzystanie tej funkcji wydaje się uzasadnione z biologicznego punktu widzenia. Ludzki mózg opiera się na przesyłaniu sygnału dopiero wtedy gdy napięcie przekroczy pewien próg [12], co w oczywisty sposób przypomina funkcję ReLU z dodanym biasem.

4.3 Normalizacja batcha

Normalizacja batcha jest metodą, która pozwala nam na budowanie głębszych sieci. Standardowym zabiegiem przy analizie danych jest ich normalizacja - pomaga nam to znajdować relacje miedzy wartościami wejściowymi. Podobna idea znajduje zastosowanie w sieciach głębokich. Ponieważ w pewien sposób możemy traktować sieci neuronowe jako wiele warstw kolejnych modeli intuicja podpowiada, że stosowanie normalizacji przed przekazaniem wyników poprzedniej wastwy do kolejnej również może być dobrym pomysłem. Okazuje się, że zastosowanie tej metody pozwala nam nie tylko budować głębsze sieci, ale także korzystać z wyższego learning rate'u, dzięki czemu sieci uczą się szybciej.

4.4 Ensembling

Ponieważ wartości sieci inicjowane są losowymi wartościami trening nie zawsze może doprowadzić nas do dokadnie takiego samego końcowego modelu. Ensembling jest ogólną ideą, która pozwala nam na wykorzystanie powyższej wady na naszą korzyść. Trenując ten sam model, na tych samych danych możemy potencjalnie otrzymać modele, które popełniają błędy w różnych miejscach, dzięki czemu licząc średnią wyników kilku z nich możemy liczyć, że nawet jeśli jakiś model popełnia błąd dla konkretnych danych to inne modele mogą ten błąd naprawić. Istotnie okazuje się, że ta metoda działa i pozwala otrzymać lepsze wyniki korzystając z tej samej architektury.
Inne podejście, które również zostało wykorzystane w tej pracy polega na uruchomieniu tego samego wytrenowanego modelu na kilku wariantach danych wejściowych. W przypadku tego konkretnego problemu wiemy, że jeśli jakiś ruch jest optymalny dla danej planszy, to po obróceniu tego ruchu i

planszy i 90 stopni wciąż ruch powinien być optymalny. Fakt ten jest wykorzystany do wykonania predykcji dla wszystkich obrotów planszy, a następnie wybraniu tego ruchu, który sieć uważa za optymalny również dla obrotów. Dodawany jest jeszcze jeden trik, polegający na próbie znalezienia odpowiedzi dla planszy z odwróconymi kolorami kamieni - idea za tym stojąca opiera się na pomyśle, że czasami łatwiej zablokować ruch przeciwnika, niż znaleźć optymalny ruch dla nas.

4.5 Adam

Adam jest algorytmem optymalizującym pewną zadaną funkcję kosztu opierającym się na *stochastic gradient descent*. *Stochastic gradient descent* różni się od czystego *gradient descent* tym, że nie optymalizuje funkcji dla wszystkich danych treningowych, tylko lokalnie dla kolejnych *batchy* danych. Główna różnica między zwykłym *stochastic gradient descent* a Adamem polega na liczeniu przesunięcia wartości nie tylko na podstawie aktualnego gradientu, ale również poprzednich. Wpływ gradientu na kolejne przesunięcia spada jednak wykładniczo. Pamiętanie poprzednich gradientów nie tylko pozwala na szybszą naukę, ale nawet osiąganie lepszych wyników przy tej samej architekturze.

5 Wyniki

Ostateczny wynik został osiągnięty przez ensembling trzech niezależnie trenowanych modeli. Poniższa tabela prezentuje skuteczność każdego z tych modeli, a także ich ensemblingu, na zbiorze testowym:

model	skuteczność
Model 1	73,12%
Model 2	72,65%
Model 3	72,95%
Ensembling	**73,34%**

6 Implementacja

Kod modelu można znaleźć pod adresem `https://github.com/SzymonStankiewicz/renju`. Model został zaimplementowany w Pythonie z wykorzystaniem biblioteki Tensorflow [1]. Struktura projektu wygląda następująco:

- data/
  - train.xml - zbiór treningowy,
  - valid.xml - zbiór walidacyjny,
  - test.xml - zbiór testowy,
- model/
  - __init__.py - budowanie całościowego modelu,
  - layers.py - implementacja warstw modelu,
- config.py - konfiguracja modelu,
- parse.py - parsowanie danych,
- runner.py - kod uruchamiający trenowanie sieci.

7 Podsumowanie

Ciężko o porównanie wyników z innymi modelami, ponieważ ten temat nie był poruszany w innych publikacjach. Jedynym możliwym odniesieniem jest wynik pierwszego etapu uczenia sieci w pracy AlphaGo, gdzie sieć zgadywała optymalne ruchy ze skutecznością 57%. Osiągnięty wynik uważam jednak za zadowalający.

Potencjalnym kolejnym krokiem po osiągnięciu tego wyniku mogłoby być wykorzystanie reinforcement learningu do znalezienia sieci, która będzie wykonywać lepsze ruchy niż profesjonalni gracze (podobnie jak zostało to zrobione w AlphaGo) a także wykorzystanie sieci jako ewaluatora w algorytmach przeszukiwania drzewa gry takich jak Monte Carlo Tree Search [3] czy Alpha-Beta Pruning [4].
Na zakończenie chciałbym podziękować Tomaszowi Wesołowskiemu za pomoc w doborze modelu i jego parametrów.

Literatura

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] L. V. Allis, H. van den Herik, and M. Huntjens. Go-moku and threat-space search. 1994.

[3] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. I. Cowling, S. Tavener, D. Perez, S. Samothrakis, S. Colton, and et al. A survey of monte carlo tree search methods. *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI*, 2012.

[4] O. E. David, N. S. Netanyahu, and L. Wolf. *DeepChess: End-to-End Deep Neural Network for Automatic Learning in Chess*, pages 88–96. Springer International Publishing, Cham, 2016.

[5] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[6] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[7] S. Hochreiter and J. Schmidhuber. Long short-term memory. 9:1735–80, 12 1997.

[8] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[9] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[11] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, jan 2016.

[12] G. Stuart, N. Spruston, B. Sakmann, and M. Häusser. Action potential initiation and backpropagation in neurons of the mammalian cns. *Trends in Neurosciences*, 20(3):125 – 131, 1997.