Zajęcia 1: Głębokie sieci neuronowe

2024-10-09

Głęboka sieć neuronowa ma dużo warstw neuronów. Tak samo jak sieci płytkie obliczają one funkcje fragmentarycznie liniowe (przy aktywacji ReLU), ale potrzebują mniej parametrów żeby wyrazić tyle samo obszarów, czyli są praktyczniejsze.

Można łączyć płytkie sieci – puszczamy jedną i na jej wyniku puszczamy drugą, już to daje więcej regionów niż płytkie sieci, a jest relatywnie proste. Połączenie dwóch sieci płytkich jest równoważne głębokiej sieci z dwoma ukrytymi warstwami, w której wszystkie neurony z sąsiadujących warstw są połączone (wynik z neuronów pierwszej warstwy przechodzi przez funkcję liniową i aktywację, tak samo jak podczas przechodzenia przez drugą sieć).

Ogólnie głęboka sieć z K warstwami (o $D_1, \ldots, D_K$ neuronach w każdej warstwie, wymiarze wejścia D_0 i wyjścia D_{K+1}) oblicza wartości na każdej warstwie:

$$h_1 = a\left[\beta_0 + \Omega_0 x\right], \quad h_2 = a\left[\beta_1 + \Omega_1 h_1\right], \ldots, h_K = a\left[\beta_{K-1} + \Omega_{K-1} h_{K-1}\right]$$

i ostatecznie wynik $y = \beta_K + \Omega_K h_K$, gdzie $\Omega_i \in \mathbb{R}^{D_{i+1} \times D_i}$ jest macierzą zawierającą wagi, która przekształca D_i-wymiarowy wektor h_i na D_{i+1}-wymiarowy wektor h_{i+1}. $\beta_i \in \mathbb{R}^{D_{i+1}}$ jest wektorem zawierającym wyrazy wolne, które są dodawane do wektora h_{i+1}.

Głęboka sieć z jednym wejściem, wyjściem i K warstwami zawierająca łącznie $D > 2$ neuronów może utworzyć $(D+1)^K$ regionów liniowych, mając $3D + 1 + (K-1)D(D+1)$ parametrów, natomiast płytka sieć z D neuronami może utworzyć $D + 1$ regionów przy $3D + 1$ parametrach. Czyli głębokie sieci potrzebują mniej parametrów do takiej samej wydajności.

Głębokie sieci generalizują lepiej niż płytkie, ale są trudniejsze do trenowania, potrzeba lepszych technik. Zwykły GD wymaga liczenia ogromnej ilości pochodnych, które mogą być bardzo skomplikowane.

Propagacja wsteczna (back-propagation). Liczenie pochodnych błędu po parametrach głębokiej sieci jest trudne, mamy mnóstwo złożonych funkcji. Zamiast tego wykorzystujemy algorytm:

- przebieg w przód (forward pass) – obliczamy i zapamiętujemy wszystkie pośrednie wyniki (wartości funkcji liniowych od kolejnych warstw i te wartości obłożone funkcją aktywacji, ostateczny wyniki i funkcja straty).
- przebieg w tył I (backward pass I) – możemy łatwo policzyć pochodną $\frac{\partial L}{\partial y}$ bezpośrednio, pochodne straty po kolejnych warstwach liczymy jako iloczyn $\frac{\partial h_{i+1}}{\partial h_i}$ (obliczamy bezpośrednio) i już wyliczonej pochodnej $\frac{\partial L}{\partial h_{i+1}}$.
- przebieg w tył II (backward pass II) – następnie w taki sam sposób liczymy pochodne straty po parametrach sieci.

Zajęcia 2: Funkcje kosztu

2024-10-23

Na nasz model $f[x, \phi]$ patrzymy jak na rozkład prawdopodobieństwa warunkowego $Pr\left(y \mid x\right)$. Funkcje straty konstruujemy tak, by dla każdej pary treningowej $\{x_i, y_i\}$ prawdopodobieństwo $Pr\left(y_i \mid x_i\right)$ było jak największe – chcemy uzyskać rozkład, którego maksymalna wartość jest równa tej właściwej.

W tym celu wybieramy sensowny dla naszego przypadku rozkład prawdopodobieństwa o pewnych parametrach Θ, co daje nam rozkład prawdopodobieństwa $Pr\left(y \mid \Theta\right)$. Nasza sieć będzie obliczała (niekoniecznie wszystkie) parametry z Θ, w ten sposób dostaniemy rozkład prawdopodobieństwa $Pr\left(y \mid f\left[x, \phi\right]\right)$. Będziemy chcieli tak wytrenować naszą sieć, żeby najbardziej prawdopodobna była wartość poprawna. Będziemy zwracać albo cały rozkład (parametry), albo najbardziej prawdopodobną wartość.

Chcemy zmaksymalizować $Pr\left(y_i \mid x_i\right)$, a więc znaleźć

$$\hat{\phi} = \arg\max_{\phi}\left[Pr\left(y_1, \ldots, y_l \mid x_1, \ldots, x_l\right)\right] = \arg\max_{\phi}\left[\prod_{i=1}^{l} Pr\left(y_i \mid f\left[x_i, \phi\right]\right)\right],$$

gdzie w tym przejściu korzystamy z założenia o niezależności naszych przewidywań.

Można patrzeć na logarytm, bo mnożenie małych liczb powoduje błędy numeryczne. Dlatego często maksymalizujemy wiarygodność logarytmiczną (log-likelihood):

$$\hat{\phi} = \arg\max_{\phi} \left[\log \left[\prod_{i=1}^{l} Pr\left(y_i \mid f\left[x_i, \phi\right]\right) \right] \right] = \arg\max_{\phi} \left[\sum_{i=1}^{l} \log \left[Pr\left(y_i \mid f\left[x_i, \phi\right]\right) \right] \right].$$

Zamiast maksymalizować wygodniej jest minimalizować, więc zazwyczaj będziemy minimalizować funkcję straty (kosztu):

$$L\left[\phi\right] = \sum_{i=1}^{l} -\log \left[Pr\left(y_i \mid f\left[x_i, \phi\right]\right) \right],$$

którą nazywamy ujemną log-wiarygodnością (entropią krzyżową).

Rozbieżność (dywergencja) Kullbacka-Leiblera (KLD, Kullback-Leibler divergence) to wartość

$$D_{KL}[q(y) \,||\, p(y)] = \int_{-\infty}^{\infty} q(y) \log \frac{q(y)}{p(y)},$$

która mierzy ilość utraconej informacji, gdy $p(y)$ przybliża $q(y)$.

Entropia różniczkowa rozkładu prawdopodobieństwa to wartość

$$H\left(q\left(y\right)\right) = \int_{-\infty}^{\infty} q\left(y\right) \log q\left(y\right).$$

KLD jest nieujemna, bo

$$\begin{aligned} D_{KL}\left[q\left(x\right) \,||\, p\left(y\right)\right] &= -\int_{-\infty}^{\infty} q\left(y\right) \log \frac{p\left(y\right)}{q\left(y\right)} \geq \int_{-\infty}^{\infty} q\left(y\right) \left(1 - \frac{p\left(y\right)}{q\left(y\right)}\right) \\ &= \int_{-\infty}^{\infty} q\left(y\right) - p\left(y\right) = 1 - 1 = 0, \end{aligned}$$

gdzie nierówność jest zastosowaniem $-\log x \geq 1 - x$. KLD nie ma innych fajnych własności, bo nie jest symetryczna ani nie spełnia nierówności trójkąta.

Zazwyczaj stosujemy $q\left(y\right) = \frac{1}{l} \sum_{i=1}^{l} \delta\left(y - y_i\right)$, gdzie δ to funkcja Diraca, która ma pole 1 i jedyną niezerową wartość dla argumentu 0.

Analogicznie jak przedtem mamy

$$\hat{\phi} = \arg\max_{\phi} \left[D_{KL}\left[q\left(y\right) \,||\, p\left(y\right)\right]\right] = \arg\min_{\phi} \left[-\sum_{i=1}^{l} \log \left[p\left(y_i \mid f\left[x_i, \phi\right]\right)\right] \right],$$

gdzie przejścia pomiędzy to podstawianie definicji i pozbywanie się wyrażeń niezależnych od ϕ.

Algorytm 1 (Konstruowanie funkcji straty)**.** Wybieramy rozkład prawdopodobieństwa $Pr\left(y \mid \Theta\right)$ sensowny dla naszego przypadku (np. Gaussa albo Bernoulliego). Następnie tworzymy model uczenia maszynowego $f\left[x, \phi\right]$, który przewiduje jeden lub kilka parametrów, więc

$$Pr\left(y \mid \Theta\right) = Pr\left(y \mid f\left[x, \phi\right]\right).$$

Stosując algorytm uczenia maszynowego minimalizujemy entropię krzyżową, mając wytrenowany model zwracamy rozkład prawdopodobieństwa lub jego maksimum.

Przykład (Regresja jednowymiarowa)**.** Wybieramy rozkład Gaussa o wartości oczekiwanej μ i wariancji σ^2. Mamy więc

$$Pr\left(y \mid \mu, \sigma\right) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{\left(y - \mu\right)^2}{2\sigma^2}\right).$$

Nasz model będzie wyznaczał $\mu = f\left[x, \phi\right]$. Po przeliczeniu entropii krzyżowej dostajemy funkcję

kosztu

$$L[\phi] = \sum_{i=1}^{l} (y - f[x, \phi])^2,$$

czyli znaną nam już kwadratową funkcję straty.

Model homoskedastyczny to taki, który zakłada, że wariancja danych jest stała. Może to być nierealne, wtedy stosuje się modele heteroskedastyczne, które przewidują również wariancję (przy rozkładzie Gaussa daje nam to dużo brzydszą funkcję kosztu, bo wszystkie składniki są zależne od parametrów).

Przykład (Klasyfikacja binarna). Wybieramy rozkład Bernoulliego z parametrem $\lambda \in (0, 1)$, czyli rozkład $Pr(y \mid \lambda) = (1-\lambda)^{1-y} \cdot \lambda^y$ ($y = 1$ ma prawdopodobieństwo λ). Wynik naszej sieci będziemy przepuszczać przez funkcje sigmoid $\sigma(x) = \frac{1}{1+\exp(-x)}$, by znormalizować wyniki do przedziału $(0, 1)$. Daje nam to funkcję straty

$$L[\phi] = \sum_{i=1}^{l} -(1-y_i) \log[1 - \sigma[f[x_i, \phi]]] - y_i \log[\sigma[f[x_i, \phi]]],$$

którą nazywamy binarną entropią krzyżową.

Przykład (Klasyfikacja wieloklasowa). Podobnie jak w klasyfikacji binarnej będziemy wyznaczać parametry $\lambda_1, \ldots, \lambda_k$, gdzie k jest liczbą klas. Te wagi będą oznaczały prawdopodobieństwo należenia wejścia do każdej z klas i powinny sumować się do 1. W tym celu wynik sieci przepuścimy przez funkcję softmax $\text{softmax}_j[x] = \frac{\exp[x_j]}{\sum_{i=1}^{k} \exp[x_i]}$, która normalizuje wektor długości k tak, by suma jego współrzędnych wynosiła 1. Daje nam to funkcję straty

$$L[\phi] = -\sum_{i=1}^{l} \left(f_{y_i}[x_i, \phi] - \log\left[\sum_{j=1}^{k} \exp[f_j[x_i, \phi]]\right]\right),$$

gdzie napis f_j oznacza wartość na j-tym indeksie wynikowego wektora. Powyższa funkcja znana jest jako wieloklasowa entropia krzyżowa (multiclass cross-entropy loss).

Przy więcej niż jednym wyjściu naszej sieci zazwyczaj traktujemy każdy wymiar jako niezależny i mamy

$$Pr(y \mid f[x_i, \phi]) = \prod_d Pr(y_d \mid f_d[x_i, \phi]),$$

gdzie d to liczba wymiarów wyjścia sieci f.

Źródłami błędu w sieci neuronowej są:

- szum (noise) – niepewność wewnętrzna problemu, która powoduje, że możemy dostać różne etykiety dla tego samego wejścia (np. gdy przybliżamy jakąś funkcję i znamy jej wartość z przybliżeniem)
- odchylenie (bias) – niezdolność modelu do dopasowania się do prawdziwej funkcji nawet dla najlepszych danych, wynika z ograniczeń wybranego modelu
- wariancja – błąd wynikający ze słabości danych treningowych (jest ich mało lub są niereprezentatywne)

Dla danych testowych, dla których szum daje danym odchylenie od średniej wartości σ mamy wartość oczekiwaną i wariancję:

$$\mu[x] = E[y[x]] = \int y[x] \, Pr(y \mid x) \, \mathrm{d}y$$

$$\sigma^2(x) = E\left[(\mu[x] - y[x])^2\right],$$

gdzie $y[x]$ to wartość y dla danego x.

Zatem stosując stratę najmniejszych kwadratów dostajemy

$$L[x] = (f[x, \phi] - y[x])^2 = ((f[x, \phi] - \mu[x]) + (\mu[x] - y[x]))^2,$$

co daje wartość oczekiwaną błędu

$$E_y\left[L\left[x\right]\right] = \left(f\left[x,\phi\right] - \mu\left[x\right]\right)^2 + \sigma^2\left[x\right].$$

Zatem poza stratą wynikającą z modelu mamy też stratę wynikającą z szumu.

Parametry modelu ϕ zależą od zbioru treningowego $D = \{x_i, y_i\}$, a więc właściwie powinniśmy pisać $f\left[x,\phi\left[D\right]\right]$. Oczekiwany model definiujemy jako

$$f_\mu\left[x\right] = E_D\left[f\left[x,\phi\left[D\right]\right]\right].$$

Rozbijając kwadratowy koszt (bez szumu) na części dostajemy:

$$\left(f\left[x,\phi\left[D\right]\right] - \mu\left[x\right]\right)^2 = \left(\left(f\left[x,\phi\left[D\right]\right] - f_\mu\left[x\right]\right) + \left(f_\mu\left[x\right] - \mu\left[x\right]\right)\right)^2.$$

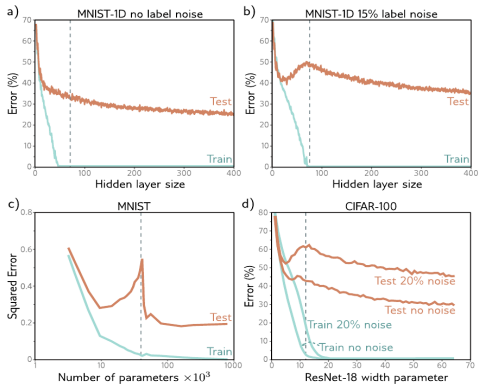
Co ostatecznie daje nam

$$E_D\left[E_y\left[L\left[x\right]\right]\right] = E_D\left[\left(f\left[x,\phi\left[D\right]\right] - f_\mu\left[x\right]\right)^2\right] + \left(f_\mu\left[x\right] - \mu\left[x\right]\right)^2 + \sigma\left[x\right]^2,$$

gdzie kolejne czynniki to odpowiednio wariancja (niepewność w kontekście zbioru treningowego), odchylenie (odchylenie modelu od modelowanej funkcji) i szum. Dla innych zadań niż regresja te trzy składniki mogą wyglądać inaczej.

Wariancję można zmniejszać zwiększając zbiór treningowy, a odchylenie zwiększając liczbę neuronów (moc sieci). Zwiększanie mocy sieci przy ustalonym zbiorze treningowym zmniejsza odchylenie, ale zwiększa wariancję (bias-variance tradeoff) – od pewnego momentu odchylenie jest zerowe (model uczy się danych na pamięć), ale przez wariancję błąd robi się coraz większy.

Zjawisko Double Descent polega na tym, że zwiększając liczbę parametrów sieci dla ustalonego zbioru treningowego najpierw widzimy spadek błędu (lepiej dopasowujemy się do funkcji), a po pewnym czasie błąd zaczyna rosnąć – wariancja robi się duża. Następnie (dla zbiorów danych ze sporym szumem) następuje kolejny spadek straty. Nie wiadomo do końca z czego wynika, teoria jest taka, że dla wielowymiarowych problemów mamy bardzo mało danych treningowych w porównaniu z możliwymi opcjami, a model (mimo idealnego dopasowania do danych) priorytetyzuje w dalszym uczeniu rozwiązania, które są bardziej regularne (priorytetyzowanie jednych rozwiązań nad inne nazywamy odchyleniem indukcyjnym – inductive bias), a takie raczej lepiej się generalizują. Nie wiemy, czemu tak jest, ale zakładamy, że regularyzatorami sieci (czynnikami promującymi regularność rozwiązań) są parametry inicjalizacji i algorytm trenujący.



Rysunek 1: Double Descent. Pierwsza część wykresu jest nazywana classical (under-parametrized) regime, a druga modern (over-parametrized) regime. Środek wykresu, gdzie błąd rośnie, to critical regime

Zajęcia 3: Trenowanie sieci

2024-10-30

Propozycja 1. Jeśli a jest zmienną losową o rozkładzie prawdopodobieństwa g symetrycznym względem 0 oraz $E[a] = 0$ i σ^2 to jej wariancja, to dla $b = ReLU(a)$ jest $E\left[b^2\right] = \frac{\sigma^2}{2}$.

Dowód.

$$E\left[b^2\right] = \int_{-\infty}^{0} g(b)\, b^2\, \mathrm{d}b + \int_{0}^{\infty} g(b)\, b^2\, \mathrm{d}b = \int_{0}^{\infty} g(a)\, a^2\, \mathrm{d}a = \frac{E\left[a^2\right]}{2} = \frac{\sigma^2}{2},$$

gdzie ostatnie przejście wynika z $E[a]^2 = 0$ a poprzednie z symetryczności g względem 0. □

Propozycja 2. Rozważmy warstwę f i kolejną zadaną wzorem $f'\beta + \Omega a[f]$. Załóżmy, że f ma wariancję σ_f^2, a wagi inicjalizujemy $\beta_i = 0$ i Ω_{ij} z rozkładu normalnego z $\mu = 0$ i wariancją σ_Ω^2. Zachodzi $E[f'] = 0$.

Dowód.

$$E[f_i'] = E\left[\beta_i + \sum_{j=1}^{D_f} \Omega_{ij} a[f_j]\right] = E[\beta_i] + \sum_{j=1}^{D_f} E[\Omega_{ij}]\, E[a[f_j]] = 0,$$

gdzie D_f oznacza wymiar warstwy f. Przedostatnie przejście wynika z niezależności wszystkich zmiennych. □

Propozycja 3. Wariancja $\sigma_{f'}^2$ wynosi $\frac{1}{2} D_f \sigma_\Omega^2 \sigma_f^2$.

Dowód.

$$\sigma_{f'}^2 = E\left[{f'}_i^2\right] - E[{f'}_i]^2 = E\left[\left(\beta_i + \sum_{j=1}^{Df} \Omega_{ij} a[f_j]\right)^2\right] - 0 = E\left[\left(\sum_{j=1}^{Df} \Omega_{ij} a[f_j]\right)^2\right]$$

$$= \sum_{j=1}^{D_f} E\left[\Omega_{ij}^2\right] E\left[a[f_j]^2\right] = \sigma_\Omega^2 \sum_{j=1}^{D_f} E\left[a[f_j]^2\right] = \frac{1}{2} D_f \sigma_\Omega^2 \sigma_f^2,$$

gdzie trzecie i czwarte przejście wynika z tego, że wartość oczekiwana każdej z wag wynosi 0. □

Algorytm 2 (Inicjalizacja He)**.** Chcemy utrzymywać wariancję kolejnych warstw na tym samym poziomie. W tym celu ustanawiamy $\sigma_\Omega^2 = \frac{2}{D_f}$ (wariancja wag zależy od rozmiaru poprzedniej warstwy).

Zauważmy, że podczas przejścia w tym w algorytmie propagacji wstecznej mnożymy przez macierz Ω^T, a więc analogiczne przejścia dla liczonych pochodnych dadzą nam optymalną wartość $\sigma_\Omega^2 = \frac{2}{D_{f'}}$. Jeśli macierz Ω nie jest kwadratowa, to nie da się spełnić obu równań, ale można wziąć średnią i przyjąć $\sigma_\Omega^2 = \frac{4}{D_f + D_{f'}}$.

Inicjalizację He (zwłaszcza pierwszą wersję) stosuje się, gdy funkcją aktywacji jest ReLU (lub jego wariant – Leaky ReLU, ELU) lub funkcje typu Swish, Mish

Algorytm 3 (Inicjalizacja Xaviera (Glorot))**.** Stosujemy rozkład normalny o wartości oczekiwanej 0 i wariancji $\sigma^2 = \frac{2}{D_f + D_{f'}}$ (odwrotność średniej liczby wymiarów) lub rozkład jednostajny z przedziału $[-r, r]$ dla $r = \sqrt{3\sigma^2}$. Zazwyczaj stosuje się przy braku aktywacji lub funkcjach typu tanh, sigmoid, softmax.

Algorytm 4 (Inicjalizacja LeCuna)**.** Stosujemy rozkład normalny o wartości oczekiwanej 0 i wariancji $\sigma^2 = \frac{1}{D_f}$ lub rozkład jednostajny z przedziału $[-r, r]$ dla $r = \sqrt{3\sigma^2}$. Stosuje się przy funkcji aktywacji SELU.

Kiedyś używano sigmoidu zamiast ReLU, ale występuje tam vanishing gradient problem (gradient jest bardzo mały dla wartości bliżej końców rozkładu), ReLU prostsze obliczeniowo.

Dying ReLU problem: w czasie treningu niektóre neurony umierają (zwracają tylko 0) – suma ważona neuronów wejścia jest ujemna, nie zmieni się to, bo pochodna ReLU jest w takiej sytuacji zawsze zerowa.

Czasem może umrzeć nawet połowa neuronów w sieci.

Definicja 1. LeakyReLU to funkcja $\text{LeakyReLU}(z) = \max(\alpha z, z)$. Jest to ReLU, które przepuszcza ujemne wartości, ale pomnożone przez małą stała. Dzięki temu martwe neurony mogą odżyć. Całkiem dobrze działa $\alpha = 0.2$. Ewentualne wariacje: Randomized ReLU (co użycie losuje się α z ustalonego podprzedziału $[0, 1]$, przy testach bierze się średnią wartość), PReLU (parametric ReLU, α zostaje wyuczona w trakcie treningu).

Definicja 2. Exponential Linear Unit (ELU) to funkcja $\text{ELU}_\alpha(z) = \begin{cases} \alpha(e^z - 1) & z < 0 \\ z & \text{wpp} \end{cases}$ dla zadanego parametru α. Taka funkcja nie nasyca się (brak vanishing gradient problem), pochodna nie jest zerowa (brak umierania neuronów), funkcja jest gładka (bardziej regularna, przyspiesza proces uczenia). Minus jest taki, że liczy się ją cięższej niż ReLU.

Definicja 3. Scaled ELU (SELU) to przeskalowana funkcja ELU, $\text{SELU}(z) = 1.05\,\text{ELU}_{1.67}(z)$. Użycie SELU dla sieci neuronowej złożonej tylko z gęstych warstw daje efekt samonormalizacji (wyjście każdej warstwy ma $\mu = 0, \sigma = 1$), ale działa to tylko, gdy wejście też jest znormalizowane, stosujemy inicjalizację LeCuna i nie używamy innych technik regularyzacji (l_1, l_2, batch-norm i inne). Są to bardzo silne warunki, ale jeśli są spełnione, to SELU jest najlepsza.

Definicja 4. Gaussian ELU (GELU) to funkcja $\text{GELU}(z) = z\Phi(z)$, gdzie $\Phi(z)$ oznacza prawdopodobieństwo, że losowa liczba ze znormalizowanego rozkładu normalnego jest mniejsza niż z. Jest gładka, niemonotoniczna (dobrze, bo może wyłapać skomplikowane zależności), działa prawdopodobnie najlepiej, ale jest kosztowna obliczeniowo. Dobrze aproksymuje są $z\sigma(1.702z)$, czyli szczególny przypadek swish.

Definicja 5. Swish to funkcja $\text{Swish}_\beta(z) = z\sigma(\beta z)$, gdzie σ to sigmoid. Parametr β może być ustalony lub uczony wraz z wagami.

Definicja 6. Mish to funkcja $\text{Mish}(z) = z\tanh(\ln(1 + e^z))$. Jest gładka, niemonotoniczna, niewypukła, podobna do swisha lub GELU.

Ogólnie wybierając funkcję aktywacji musimy kierować się poziomem skomplikowania zadania i wymaganą prędkością obliczeń. ReLU działa dobrze dla prostych zadań, a hardware i biblioteki dobrze ją optymalizują. Przy cięższy zadaniu można stosować leaky ReLU, jeśli czas jest ważny. Jeśli nie jest, to dobre wyniki daje swish lub mish (ale wymaga dużo obliczeń). Dla funkcji spełniających odpowiednie wymagania najlepsze jest SELU.

Algorytm 5 (Batch Normalization). Mimo zastosowania He z ReLU problemy z gradientem mogą pojawiać się w późniejszych fazach treningu, stosujemy normalizację batcha, czyli dla całego batcha (co warstwę przed lub po aktywacji) obliczamy jego średnią, następnie wariancję, normalizujemy wektory z batcha i wyliczamy wynik warstwy (mnożymy przez wagi i dodajemy bias). Opisują to wzory:

$$\mu_B = \frac{1}{m_B}\sum_{i=1}^{m_B} x^{(i)}$$

$$\sigma_B^2 = \frac{1}{m_B}\sum_{i=1}^{m_B}\left(x^{(i)} - \mu_B\right)^2$$

$$\hat{x}^{(i)} = \frac{x^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

$$z^{(i)} = \gamma\hat{x}^{(i)} + \beta.$$

Gdzie $x^{(i)}$ to i-ty wektor z batcha (rozmiaru m_B), ε dodajemy, żeby nie dzielić przez 0, a γ, β są wektorami rozmiaru takiego jak $x^{(i)}$ (mnożymy pola wektora punktowo), których się uczymy

podczas treningu.

Jest to technika regularyzacji, która ogranicza potrzebę stosowania innych. Zwiększa złożoność, choć przyspiesza zbieżność, ale daje bardzo dobre wyniki. Jeśli poprzednia warstwa oblicza $XW + b$, to można zastosować BN "w locie" licząc od razu jej wynik

$$\gamma \frac{XW + b\mu}{\sigma + \beta},$$

albo przyjmując $W' = \frac{\gamma W}{\sigma + b}$ i $b' = \gamma (b - \mu) \sigma + \beta$ mamy wynik

$$XW' + b'.$$

Algorytm 6 (Gradient clipping)**.** Technika przycinania każdego gradientu do przedziału $[-1, 1]$, która pomaga na exploding gradient problem. Używana jest głównie przy rekurencyjnych sieciach neuronowych.

Algorytm 7 (Transfer learning)**.** Mając już wytrenowaną sieć działającą dobrze dla jakiegoś zadania można wziąć jej wytrenowane warstwy i wykorzystać do innego, podobnego zadania. Pierwsze warstwy sieci zazwyczaj uczą się cech ogólnych, one będą dla nas wartościowe. Do wytrenowanych warstw dokładamy nowe i je trenujemy, na koniec dostrajamy wszystkie warstwy. Czasem daje to dobre wyniki, ale nie działa dobrze dla małych, gęstych sieci, bo małe sieci nie uczą się wielu wzorców, a gęstość priorytetyzuje specyficzne wzorce, nieużyteczne gdzie indziej. Najlepiej sprawdza się w głębokich sieciach konwolucyjnych.

Algorytm 8 (Stochastic Gradient Descent)**.** Funkcja może mieć wiele minimów lokalnych, na których zwykły gradient descent się zatrzyma. Dlatego wprowadzamy losowość: wybieramy losowy batch (zazwyczaj bez zwracania, kończymy epokę, gdy skończą się nam dane) i liczymy gradient dla każdego wejścia z batcha, a następnie na raz aktualizujemy wagi. To znaczy, robimy

$$\phi_{t+1} = \phi_t - \alpha \sum_{i \in B_t} \frac{\partial \ell_i[\phi_t]}{\partial \phi}$$

zamiast aktualizować wagi po przeliczeniu każdego wejścia. Dobrze działa z learning rate schedule: zaczynamy z dużym α, co iterację dzielimy przez stałą wartość.

Dzięki wprowadzeniu losowości nasz algorytm jest w stanie wyjść z minimów lokalnych.

Algorytm 9 (SGD z pędem)**.** Robimy SGD, ale aktualizujemy parametry o wartość będącą ważoną kombinacją gradientu aktualnego batcha i poprzedniej zmiany. Wyznaczamy:

$$m_{t+1} = \beta m_t + (1 - \beta) \sum_{i \in B_t} \frac{\partial \ell_i [\phi_t]}{\partial \phi}$$

$$\phi_{t+1} = \phi_t - \alpha m_{t+1}$$

dla pewnego β (momentum). Szybkość uczenia wzrasta, jeśli gradienty mają podobny kierunek, inaczej składniki sumy się znoszą. Ogólnie taki algorytm powoduje, że trajektoria zmian jest gładsza i algorytm mniej kluczy.

Algorytm 10 (Nesterov Accelerated Momentum)**.** Jak SGD z pędem, ale zauważamy, że pojęcie pędu oznacza, gdzie z grubsza algorytm będzie się dalej poruszać. Przyspieszony pęd Nesterova oblicza gradienty w tym przewidywanym punkcie.

$$m_{t+1} = \beta m_t + (1 - \beta) \sum_{i \in B_t} \frac{\partial \ell_i [\phi_t - \alpha \beta m_t]}{\partial \phi}$$

$$\phi_{t+1} = \phi_t - \alpha m_{t+1}$$

Algorytm 11 (Znormalizowany GD). Strata może zmieniać się szybko w jednym wymiarze, a wolno w drugim. Mały learning rate spowoduje wolne zbliżanie się do minimum w drugim wymiarze, ale duży spowoduje destabilizację w pierwszym. Rozwiązaniem jest normalizowanie gradientu do przedziału $[-1, 1]$:

$$m_{t+1} = \frac{\partial L\left[\phi_t\right]}{\partial \phi}$$

$$v_{t+1} = m_{t+1}^2$$

$$\phi_{t+1} = \phi_t - \alpha \frac{m_{t+1}}{\sqrt{v_{t+1}} + \varepsilon}$$

gdzie mnożenie w drugim punkcie dzieje się po osiach (czyli jakbyśmy liczyli iloczyn skalarny).

Algorytm 12 (AdaGrad). Adaptive gradient, działa podobnie jak normalizowanie GD. Liczymy sumę kwadratów gradientów, a następnie przez nią normalizujemy:

$$s_{t+1} = s_t + m_{t+1}^2$$

$$\phi_{t+1} = \phi_t - \alpha \frac{m_{t+1}}{\sqrt{s} + \varepsilon}$$

Normalizuje to gradient, ale powoduje, że staje się on coraz mniejszy, więc uczenie może zanikać przedwcześnie (zwłaszcza dla skomplikowanych problemów).

Algorytm 13 (RMSProp). Działa jak AdaGrad, ale kumuluje kwadrat gradientu jako kombinację ważoną poprzedniego i aktualnego:

$$s_{t+1} = \rho s + (1 - \rho)\, m_{t+1}^2.$$

Dzięki temu uczenie nie spowalnia.

Algorytm 14 (Adaptive moment estimation (Adam)). Znormalizowany SGD z pędem. Liczymy

$$m_{t+1} = \beta m_t + (1 - \beta)\, \frac{\partial L\left[\phi_t\right]}{\partial \phi}$$

$$v_{t+1} = \gamma v_t + (1 - \gamma) \left(\frac{\partial L\left[\phi_t\right]}{\partial \phi} \right)^2$$

Te wartości są początkowo małe, więc ustalamy:

$$\widehat{m_{t+1}} = \frac{m_{t+1}}{1 - \beta^{t+1}}$$

$$\widehat{v_{t+1}} = \frac{v_{t+1}}{1 - \gamma^{t+1}},$$

które dla późniejszych iteracji zbiegają do m_{t+1} i v_{t+1}. Ostatecznie aktualizujemy:

$$\phi_{t+1} = \phi_t - \alpha \frac{\widehat{m_{t+1}}}{\sqrt{\widehat{v_{t+1}}} + \varepsilon}.$$

Zazwyczaj przyjmujemy dość duże parametry ($\beta = 0.9$, $\gamma = 0.999$).

Adam ma różne warianty, przede wszystkim NAdam (Nesterov Adam) używający pędu Nesterova oraz AdaMax, gdzie v_{t+1} powstaje poprzez wzór

$$v_{t+1} = \max\left(\gamma v_t, \left\| \frac{\partial L\left[\phi_t\right]}{\partial \phi} \right\| \right).$$

Algorytm 15 (Weight decay). Zmieniając wagi podczas GD można przemnożyć aktualne wagi przez

stałą mniejszą od 1 i dopiero uwzględnić gradient. Jest to technika regularyzacji, która ma ograniczać wielkość wag.

$$\phi_{t+1} = (1-\lambda)\,\phi_t - \alpha\frac{\partial L\left[\phi_t\right]}{\partial \phi}.$$

Uwaga. Weight decay dla zwykłego SGD jest równoważny regularyzacji L2 z pewnym parametrem. Regularyzacja L2 (Tichonowa) polega na dodaniu do minimalizowanej wielkości (błędu) pewnej wielokrotności rozmiaru argumentu (wag).

Algorytm 16 (AdamW). Zwykły Adam źle działa z regularyzacją L2, zamiast tego stosuje się (nierównoważny w przypadku Adama) weight decay.

Zajęcia 4: Regularyzacja

2024-11-06

Jeśli współczynnik uczenia się jest za mały, to nie dotrzemy do minimum, jak będzie za duży, to będziemy krążyć wokół minimum, nie docierając do niego.

Różne pomysły (planery):

- constant, zaczynanie z bardzo małym α, zwiększanie go stopniowo o ustaloną stałą multiplikatywną. Wartość straty względem α będzie maleć, gdy zwiększamy α, aż zacznie rosnąć. Wybieramy α tak, by było 10 razy mniejsze niż wartość, gdzie strata zaczyna rosnąć.
- power scheduling, czyli ustalamy
$$\alpha\left(t\right) = \frac{\alpha\left(0\right)}{\left(1+\frac{t}{s}\right)^c},$$
gdzie c, s to hiperparametry i zazwyczaj $c = 1$.
- exponential scheduling, zmniejszamy o zadaną stałą multiplikatywną co zadaną ilość kroków
- piecewise constant scheduling, używamy stałego α, po pewnej liczbie epok zmieniamy na inną stałą
- performance scheduling, co N kroków sprawdzamy błąd walidacyjny i dzielimy α przez ustalone λ jeśli błąd przestaje spadać
- 1cycle scheduling, przez pół treningu zwiększamy liniowo od α_0 do α_1, przez drugie pół zmniejszamy od α_1 do α_0 pomniejszonego o kilka rzędów wielkości. α_1 ustalamy jak wyżej (10 razy mniej niż punkt przegięcia), α_0 powinno być 10 razy mniejsze niż α_1.

Regularyzacja to różne techniki pozwalające na zmniejszenie błędu uogólnienia. Regularyzacja jawna polega na dodaniu do funkcji straty składnika addytywnego, który w pożądany sposób zmienia funkcję straty (np. regularyzacja $L2$). Regularyzacja niejawna jest własnością wewnętrzną algorytmów (S)GD. Regularyzacja to też inne heurystyki zwiększające wydajność modelu (early stopping, ensembling, dropout).

W regularyzacja jawnej minimalizujemy argumenty z dodatkowym czynnikiem:

$$\hat{\phi} = \arg\min_{\phi}\left[\sum_{i=1}^{I}\ell_i\left[x_i, y_i\right] + \lambda\cdot g\left(\phi\right)\right],$$

gdzie $g\left(\phi\right)$ to funkcja zwracająca skalar, który ma dużą wartość dla niepreferowanych parametrów. Taka funkcja ma inne minima niż sama funkcja straty, więc algorytm zbiega do innych parametrów.

Algorytm 17 (Regularyzacja L2). Minimalizujemy wartość

$$\hat{\phi} = \arg\min_{\phi}\left[\sum_{i=1}^{I}\ell_i\left[x_i, y_i\right] + \lambda\sum_j \phi_j^2\right].$$

Zazwyczaj stosuje się ją do wag, ale nie do biasów. Zmniejsza przetrenowanie, wagi są mniejsze, więc wyjście sieci jest mniej zróżnicowane a tworzona funkcja gładsza.

Algorytm GD zbiega do minimum inaczej, niż zrobiłby to proces ciągły. Jego dyskretna natura powoduje, że tak naprawdę jest minimalizowana funkcja

$$L_{GD}[\phi] = L[\phi] + \frac{\alpha}{4}\left\|\frac{\partial L}{\partial \phi}\right\|^2,$$

która ma takie same minima, ale inną trajektorię optymalizacji niż proces ciągły. Dlatego algorytm GD może zbiegać do innego minimum. Wynika to z faktu, że dyskretny proces jest bardziej odpychany z miejsc, gdzie jest duży gradient. Ta regularyzacja może być powodem, dla którego GD uogólnia lepiej z dużym α.

Algorytm SGD, podobnie jak GD, wprowadza niejawną regularyzację, która powoduje, że minimalizowana jest funkcja straty

$$L_{SGD}[\phi] = L[\phi] + \frac{\alpha}{4}\left\|\frac{\partial L}{\partial \phi}\right\|^2 + \frac{\alpha}{4B}\sum_{b=1}^{B}\left\|\frac{\partial L_b}{\partial \phi} - \frac{\partial L}{\partial \phi}\right\|^2,$$

gdzie L_b to średnia strata na batchu, a L to średnia strata na całym zbiorze. SGD faworyzuje trajektorie ze stabilnymi gradientami – dodatkowy wyraz odpowiada wariancji gradientów strat L_b.

Rozwiązania znalezione przez SGD generalizują się lepiej niż GD, co może wynikać z dwóch rzeczy:

- losowość SGD powoduje, że możemy uciec z lokalnych minimów
- niejawna regularyzacja promuje rozwiązania, w których wszystkie dane są dobrze dopasowane (każdy batch ma małe odchylenie od średniej w swojej stracie)

Algorytm 18 (Early stopping)**.** Trenując sieć neuronową zatrzymujemy się, gdy błąd treningowy osiągnie ustaloną wartość. Dzięki temu wagi nie staną się za duże i unikniemy przetrenowania, które powoduje, że rozwiązanie gorzej się uogólnia.

Algorytm 19 (Ensembling)**.** Korzystamy z kilku różnych modeli, dokonujemy predykcji uśredniając ich wyniki. Zakładamy, że błędy modeli są niezależne – wtedy będą się znosić.

Dwa główne pomysły to wybieranie innych inicjalizacji dla naszych modeli oraz generowanie różnych datasetów poprzez losowanie ze zwracaniem (możemy dużo razy wziąć ten sam element, bootstrap aggregating – bagging). Pierwszy pomysł w szczególności pomaga w przybliżaniu funkcji w obszarze, dla którego nie mamy danych. Inne pomysły to manipulowanie hiperparametrami lub trenowanie różnych klas modeli.

Algorytm 20 (Dropout)**.** Podczas trenowania sieci każdy neuron ma szansę (w okolicy $20-50\%$) na zostanie wyłączonym, dzięki temu sieć będzie mniej zależna od pojedynczych neuronów i wagi będą mniejsze. Pomaga wyeliminować sytuacje, w których funkcja ostro odbija od przybliżanej funkcji i potem wraca (które mogą się zdarzyć, jeśli nie ma na danym obszarze danych).

Podczas dokonywania predykcji można mnożyć wagi przez prawdopodobieństwo tego, że neuron będzie włączony. Można też stosować Monte Carlo dropout: uruchamiamy sieć wielokrotnie z różnymi wyzerowanymi neuronami i uśredniamy wynik.

Algorytm 21 (Max Norm)**.** Dla każdego neuronu oblicza się normę jego wektora wag wchodzących $\|w\|_2$ i jeśli jest większa od ustalonego r, to skaluje się go do takiej długości:

$$w = \frac{w}{\|w\|_2} \cdot r.$$

Algorytm 22 (Applying noise)**.** Możemy wygładzić wyuczoną funkcję poprzez dodanie szumu do danych, czyli wprowadzenie różnych (podobnych) etykiet dla tego samego wejścia. W przypadku problemów klasyfikacji wartości neuronów odpowiadających poprawnej klasie są dużo większe niż pozostałych (co może być problemem w uogólnianiu), więc można stosować wygładzanie etykiet – ustalony procent $\rho\%$ poprawnych etykiet zamieniamy na losowe inne. Można też zmodyfikować entropię krzyżową, by poprawna etykieta miała prawdopodobieństwo $1-\rho$, a pozostałe po tyle

> samo.

W przypadku małej ilości danych treningowych można pomóc sobie innymi danymi:

- transfer learning, czyli korzystanie z modeli wytrenowanych na pokrewnym zadaniu, zazwyczaj polega na podmienieniu kilku ostatnich warstw
- multi-task learning, czyli trenowanie sieci równolegle do kilku podobnych zadań
- self-supervised learning, czyli generowanie danych przez naszą sieć. Może to być generatywne samonadzorowane uczenie się (jednocześnie uczymy się uzupełniać dostępne dane i rozwiązywać oryginalny problem dla tych danych) lub kontrastowe nadzorowane uczenie się (pary instancji o cechach wspólnych porównuje się z parami niepowiązanymi)
- augmentation, czyli tworzenie nowych danych poprzez przerabianie tych dostępnych (np. odbijanie obrazu)