1 Problem

Chcemy na podstawie zawczasu znanego zbioru danych $X \in \mathbb{R}^{n \times d}$, i odpowiadających nim etykiet y, przewidywać etykiety $\hat{y}(x)$ dla nowych punktów $x \notin X$.

Standardowo, w problemach klasyfikacji, dzielimy dzielimy dane na zbiór treningowy, walidacyjny oraz testowy. Trenujemy na treningowym, dobieramy hiperparametry na walidacyjnym, a potem ewaluujemy na testowym. Bardziej niezawodnym mechanizmem ewaluacji jest walidacja krzyżowa (cross-validation), gdzie w kolejnych iteracjach wybieramy różne podzbiory danych jako zbiór treningowy i walidacyjny.

2 Nearest Neighbor

1. Zapamiętujemy wszystkie X i y
2. Przewidujemy klasę $\hat{y}(x)$ dla $x \notin X$, taką, że dla danej metryki odległości d:

$$\hat{y}(x) = \arg\min_{y_i \in Y} d(x, X_i)$$

Aby przeciwdziałać szumowi w danych, można brać próbkę $\{y_1, y_2, \ldots, y_k\}$, k najbliższych sąsiadów, a potem brać średnią z tych klas.

Można używać różnych d, np:

- $d(a,b) = l_1(a,b) = \sum_{i=1}^{n} |a_i - b_i|$
- $d(a,b) = l_2(a,b) = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2}$

Problemem jest złożoność obliczeniowa, naiwna wersja algorytmu musi pamiętać wszystkie X i y w pamięci, a potem dla każdego nowego punktu x musi przeszukiwać wszystkie X.

3 Regresja Liniowa

Modele nearest neighbor dynamicznie określają granicę między punktami, natomiast regresja liniowa stara się znaleźć prostą, która najlepiej rozdziela etykiety y na podstawie X.

Dla każdej klasy y_i, definiujemy wektor wag $W_i \in \mathbb{R}^d$, i bias $b_i \in \mathbb{R}$. Potem, określamy funkcję `score`:

$$f_i(x, W, b) = W_i \cdot x + b_i \quad \hat{y}(x) = \arg\max_{y_i} f_i(x, W, b)$$

Problemem dla klasyfikatorów tego typu, oczywiście, jest kształt danych. W zależności, od przyjętej przestrzeni $\mathbb{R}^d$, dane mogą być liniowo separowalne, ale w ogólności nie są.

4 Loss i regularyzacja

Aby określić jakość predykcji, dla danej funkcji $f_i(x, W, b)$, definiujemy funkcję straty (loss):

$$L(W, b) = \frac{1}{n} \sum_{i=1}^{n} L_i(W, b)$$

czyli średnia straty dla wszystkich próbek.

Aby zapobiec overfittingu, stosuje się regularyzację. W regularyzacji, dodajemy do L, karę, która jest proporcjonalna do *skomplikowania* modelu. W ten sposób model jest nagradzany za generalizowanie.

$$L(W, b) = \frac{1}{n} \sum_{i=1}^{n} L_i(W, b) + \lambda R(W)$$

Jest wiele potencjalnych mechanizmów regularyzacji:

- $R_2(W) = \sum_k \sum_l W_{k,l}^2$ - L2

- $R_1(W) = \sum_k \sum_l |W_{k,l}|$ - L1
- $R_{1+2}(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$ - L1 + L2 (Elastic net)
- Dropout
- Batch norm
- ...

Przykład 4.1

Wyobraźmy sobie, dwie różne macierze W_1 i W_2, do modelu liniowego. Mają one rozróżnić x_0 i x_1.

$$x_0 = \begin{bmatrix}1\\1\\1\\1\end{bmatrix} \quad x_1 = \begin{bmatrix}1\\0\\0\\0\end{bmatrix} \quad W_0^T = \begin{bmatrix}0\\0\\0\\1\end{bmatrix} \quad W_1^T = \frac{1}{3}\begin{bmatrix}0\\1\\1\\1\end{bmatrix}$$

$$W_0 x_0 = 1 = W_1 x_0 \quad \neq \quad W_0 x_1 = 0 = W_1 x_1$$

Lecz, regularyzacja L2 $R_2(W)$, potrafi daje inne wyniki dla W_0 i W_1:

$$R_2(W_0) = 1 \quad \neq \quad R_2(W_1) = 3 \cdot \frac{1}{9} = \frac{1}{3}$$

Więc, przy zastosowaniu regularyzacji L2, W_0 i W_1 nie są równoważne, i podczas optymalizacji, algorytm będzie preferował W_1, ponieważ ma mniejszą wartość regularyzacji.

4.1 SVM

Wieloklasowy SVM L_i, dla próbki (x_i, y_i) ma postać:

$$L_i(W, b) = \sum_{j \neq i} \max(0, f_j(x_i, W, b) - f_i(x_i, W, b) + 1)$$

Intuicyjnie, jest to ograniczona przez 0 od dołu suma różnic, między `score` każdej innej klasy, a docelowej klasy.

4.2 Softmax loss

$$L_i(W, b) = -\log P(y = i|x) = -\log\left(\frac{e^{f_i(x,W,b)}}{\sum_j e^{f_j(x,W,b)}}\right)$$

5 Optymalizacja

$$W = \arg\min_w L(w)$$

5.1 Gradient Descent

Dla t-tego kroku optymalizacji:

$$W_t = W_{t-1} - \alpha \nabla L(W_{t-1})$$

realizujemy ten krok, aż nie napotkamy na warunek stopu ($\nabla L \approx 0$), lub osiągniemy maksymalną liczbę iteracji. Oczywiście, trzeba jakoś określić W_0, czyli punkt startowy optymalizacji, tutaj zazwyczaj jest to losowa wartość. Kluczowym, z kolei, jest α, czyli *learning rate*. Tutaj jest to zazwyczaj *problem-specific*.

5.1.1 Batch GD

W Batch GD, aby zapobiec obliczaniu całkowitej funkcji straty, co jest problematyczne dla dużych zbiorów danych, obliczamy funkcję straty na całym zbiorze danych w każdej iteracji.

5.1.2 Stochastic GD

W Stochastic GD, na odwrót, obliczamy funkcję straty tylko dla jednej losowo wybranej próbki w każdej iteracji.

5.2 Momentum

Jeśli, założymy jakiś poziom racjonalności problemu optymalizacji, to przydatnym może być zastosowanie *momentum*. Momentum v jest zdefiniowane jak pęd fizyczny;

$$v_t = \rho v_{t-1} + \nabla L(W_{t-1})$$

W interpretacji fizycznej, v jest prędkością, a ρ współczynnikiem tłumienia - tarciem.

Dla t-tego kroku optymalizacji:

$$W_t = W_{t-1} - \alpha v_t$$

5.3 AdaGrad

Alternatywnym mechanizmem jest zachowywanie historycznej sumy kwadratów gradientów i używanie jej do skalowania współczynnika uczenia α.

$$\nabla_t^2 = \sum_{i=1}^{t} \nabla L(W_i)^2$$

$$W_t = W_{t-1} - \frac{\alpha}{\sqrt{\nabla_t^2 + \epsilon}} \cdot \nabla L(W_{t-1})$$

5.4 RMSProp

Jest to modyfikacja AdaGrad, która wprowadza *decay rate* w obliczaniu ∇_t^2.

$$\nabla_t^2 = \sum_{i=1}^{t} \rho \nabla_i^2 + (1-\rho) \cdot \nabla L(W_i)^2$$

5.5 Adam

Adam to połączenie *momentum* i *RMSProp*. Pamięta on dwa pierwsze momenty ruchu: v - prędkość, m - moment. Aby zapobiec problemu z $v_0 = 0, m_0 = 0$, który może wystąpić w pierwszej iteracji, stosuje się *bias correction* v_t' i m_t'.

$$v_t = \beta_1 m_{t-1} + (1-\beta_1) \cdot \nabla L(W_{t-1}) \quad v_t' = \frac{v_t}{1-\beta_1^t}$$

$$m_t = \beta_2 m_{t-1} + (1-\beta_2) \cdot \nabla L(W_{t-1})^2 \quad m_t' = \frac{m_t}{1-\beta_2^t}$$

$$W_t = W_{t-1} - \frac{\alpha v_t'}{\sqrt{m_t' + \epsilon}}$$

6 Sieci neuronowe

Zamiast stosować pojedyncze modele liniowe, stosujemy je w k warstwach, gdzie każda warstwa się składa z j neuronów. Każdy neuron w warstwie l ma wejście z warstwy $l-1$ i wyjście do warstwy $l+1$.

Dla 3-warstwowej sieci neuronowej, o funkcji aktywacji σ:

$$f(x) = W_3 \sigma(W_2 \sigma(W_1 \cdot x))$$

6.1 Funkcje aktywacji

Można stosować wielu różnych funkcji aktywacji:

- *ReLU*: $f(x) = \max(0, x)$
- *Sigmoid*: $f(x) = \frac{1}{1+e^{-x}}$
- *Tanh*: $f(x) = \tanh(x)$
- *Leaky ReLU*: $f(x) = \max(0.01x, x)$
- ...

Pozwalają one, na transformacje danych wejściowych, co ma pozwolić na liniową separowalność danych.

6.2 Konwolucja

Warstwy konwolucyjne są używane do ekstrakcji cech z danych wejściowych, takich jak obrazy. Dana warstwa konwolucyjna, ma do czynienia z pewną warstwą danych wejściowych, oraz filtrem, którego się uczy, aby wyodrębnić cechy z danych wejściowych. Hiperparametrem warstwy konwolucyjnej jest rozmiar filtra, oraz stride, który określa, co ile pikseli przesuwać filtr po obrazie.

Przykład 6.1

Dla filtra 3×3, i stride 1:

$$\begin{bmatrix} 3 & 0 & 1 & 2 \\ 1 & 5 & 8 & 9 \\ 2 & 7 & 2 & 5 \\ 0 & 1 & 3 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} -5 & -4 \\ -10 & -2 \end{bmatrix}$$

$$3 + 1 + 2 - 1 - 8 - 2 = -5$$

6.3 Pooling

Warstwy pooling są używane do zmniejszenia rozmiaru danych wejściowych, takich jak obrazy. Działają one na danych wyjściowych warstwy konwolucyjnej, redukując ich rozmiar poprzez wybieranie maksymalnych wartości w podprzedziałach. Dla danego *stride*, *kernel size* i danych wejściowych, warstwa pooling nakłada pewną prostą operację na danych.

Przykład 6.2

Dla `MaxPool` 2×2 i stride 2:

$$\begin{bmatrix} 3 & 0 & 1 & 2 \\ 1 & 5 & 8 & 9 \\ 2 & 7 & 2 & 5 \\ 0 & 1 & 3 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 5 & 9 \\ 7 & 5 \end{bmatrix}$$

6.4 Residual Networks

Im głębsze sieci są, tym większy mają problem z propagacją gradientu. Na gradient, największy wpływ ma ostatnia warstwa. Im głębiej, tym mniejszy jest wpływ na wyjście, aż w końcu, po ≈ 10 warstwach, pierwsza warstwa ma wpływ mniejszy niż dokładność `float32`.

Rozwiązaniem tego problemu jest użycie *residual connections*, które pozwalają propagować gradient przez głębokie sieci, nawet gdy gradient jest bardzo mały w ostatniej warstwie.

Dla warstwy $F_i(x)$, zdefiniujmy:

$$F_i(x) = F_{i-1}(x) + x$$

W pewnym sensie zatem, sieć się uczy odwzorowywać innej funkcji, ale w klasyfikacji wewnętrzny stan numeryczny sieci nas nie interesuje.

7 Wizja komputerowa

Dotychczas rozważaliśmy tylko klasyfikację. Co jeśli nas interesuje konkretnie pozycja obiektu na obrazie? Jest to problem regresji, a nie klasyfikacji. Tutaj, musimy traktować jako wyjście kilka mniejszych zmiennych regresyjnych, każda odpowiadająca za jedną współrzędną: $p = (x, y, w, h)$.

Powszechnym, w abstrakcyjnym kontekście detekcji obrazów, jest pre-trenowany transformer, który się składa tylko z warstw konwolucyjnych, który potem jest łączony z konkretną architekturą detekcji, w zależności od zadania.

7.1 Region-based

Aby zrealizować region-based detekcję, czyli detekcje konkretnej pozycji na obrazie, często stosowana jest region-based detekcja. Naiwną realizacją tej ideii, jest aplikowanie modelu klasyfikacji, dla wszystkich pod-obrazów w obrazie, a następnie wybór pod-obrazu, dla którego model zwrócił najwyższy wynik. Bardziej zaawansowane podejścia zazwyczaj stosują heurystyczny wybór regionów, które mogą nas interesować.

7.2 IoU

Aby porównać wynik modelu $\hat{p}$ z rzeczywistą pozycją obiektu p, często stouje się IoU (Intersection over Union), lub index Jaccarda.

$$IoU(p, \hat{p}) = \frac{p \cap \hat{p}}{p \cup \hat{p}}$$

7.3 NMS

Aby wyeliminować nadmiarowe detekcje, stosuje się NMS (Non-Maximum Suppression).

1. Wybieramy $\hat{p}' = \arg\max_{\hat{p}} IoU(p, \hat{p})$
2. Eliminujemy wszystkie $\hat{p}$, dla których $IoU(\hat{p}', \hat{p}) > \theta$
3. Wracamy do kroku 1

Intuicyjnie, dla każdego optymalnego $\hat{p}'$, eliminujemy wszystkie zbyt podobne, lecz gorsze, predykcje, aż zostają tylko optymalne wyspy, zbyt różne aby odrzucić.

7.4 mAP

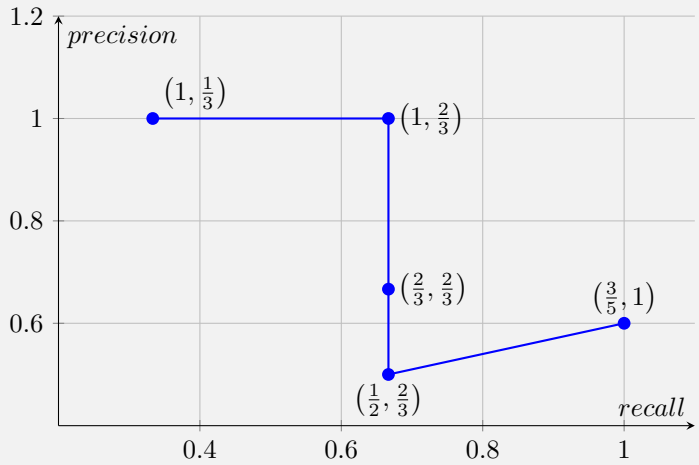
mAP (mean Average Precision) jest miarą oceny modelu klasyfikacji, która względem IoU (Intersection over Union) określa, jak dobrze model odpowiada na dane testowe.

1. Stosujemy NMS
2. Dla każdej kategorii, obliczamy Average Precision (AP), zdefiniowane jako powierzchnia pod krzywą Precision-Recall (PR) dla tej kategorii
   (a) Dla każdego przewidzianego punktu $\hat{p}$
   (b) Jeśli pokrywa się z jakimś rzeczywistym punktem p z $IoU(\hat{p}, p) > \theta$, oznacz go jako pozytywny
   (c) W p.p. oznacz go jako negatywny
   (d) Dodaj punkt do PR curve, z aktualnym stanem precyzji (% $\hat{p}$ pozytywnych) i czułości (% p z przyporządkowanym $\hat{p}$)
3. AP dla kategorii to powierzchnia pod krzywą PR dla tej kategorii
4. mAP to średnia AP dla wszystkich kategorii

Intuicyjnie, dla danej kategorii, szukamy, czy każda predykcja ma match w ground truth, mierzymy ile predykcji ma, oraz jakość predykcji. Nasz performance w czasie mierzymy, poprzez powierzchnię pod krzywą PR, i to nam daje AP. Potem uśredniamy AP dla wszystkich kategorii, aby uzyskać mAP.

Przykład 7.1

Uruchamiamy klasyfikator na różnych regionach, to nam zwraca sporo predykcji. Eliminujemy je przy pomocy NMS. Mamy pięć predykcji $\hat{p}_i$ i trzy rzeczywiste punkty p_j.
Zaczynając od punktów z największą wewnętrzną pewnością, szukamy odpowiadającego mu punktu p. Jest taki, to możemy go usunąć. Aktualnie mamy prezycję $\frac{1}{1}$, oraz czułość $\frac{1}{3}$. Dodajemy punkt do krzywej i kotynuujemy.
...
Na koniec zostaje nam ciąg punktów krzywej PR: $\left(1, \frac{1}{3}\right), \left(\frac{2}{2}, \frac{2}{3}\right), \left(\frac{2}{3}, \frac{2}{3}\right), \left(\frac{2}{4}, \frac{2}{3}\right), \left(\frac{3}{5}, \frac{3}{3}\right)$.



Powierzchnia pod tą krzywą to nasz AP.

7.5 Fast R-CNN

Zamiast odpalać nasz klasyfikator, tyle razy, ile mamy regionów, to odpalamy nasz *backbone* feature extractor, raz. Na tej przestrzeni potem odnajdujemy regiony i robimy klasyfikację. W ten sposób, część obliczeń jest wykonywana tylko raz.

Tutaj, jedynym problemem jest cropping czynników. Dostaliśmy z naszej heurystyki regionów jakieś croppy, a potrzebujemy je przenieść na przestrzeń czynników, lecz nasza wyekstraktowana przestrzeń cech nie rzutuje się 1:1 na obrazek wejściowy.

7.5.1 Rol Pool

1. Rzutujemy obrazek na przestrzeń cech
2. Niedokładne rzutowanie rozwiązujemy przez zaokrąglenie; "*snap*"
3. Dzielimy rzutowanie na grid 2×2 regionów
4. W ramach regionów robimy *max pooling*

Max pooling na koniec zapewnia, że niezależnie od rozmiaru regionu, zawsze mamy ten sam rozmiar wyjściowy. Problemem, jest samo zaokrąglanie, które dodaje niestabilność do uczenia i predykcji, szczególnie w dziwnych miejscach.

7.5.2 Rol Align

Jest to bardzo podobna metoda to *Rol Pool*, lecz, zamiast zaokrąglenia, wykorzystuje interpolację. Jak dostajemy punkt regionu między czynnikami przestrzeni czynników, definujemy nowy czynnik, jako kombinację liniową czynników przestrzeni, w taki sposób, że punkt regionu jest w środku nowego czynnika.

Intuicyjnie, zamiast zaokrąglenia, interpolujemy i uśredniamy różnice między regionami na obrazku, a przestrzenią czynników.

7.6 RPN

Zamiast heurystycznie szukać regionów, do konkretnego problemu, możemy nauczyć sień neuronową, aby to robiła za nas.

8 Segmentacja Semantyczna

W semantycznej segmentacji, każdemu pikselowi chcemy przyporządkować pewną klasę semantyczną w obrazku. Naiwnym rozwiązaniem jest, po raz kolejny, przesuwanie okienka kontekstowego i aplikowanie modelu klasyfikacyjnego na każdym pikselu.

8.1 U-net

Bardziej kompleksowym rozwiązaniem jest pełna konwolucyjna sieć neuronowa (CNN), która jest trenowana na całym obrazie i każdemu pikselowi przyporządkowuje klasę semantyczną. Problemem, jest ograniczone *receptive field* każdej warstwy konwolucyjnej, oraz koszta związane z tym podejściem.

Tutaj, standardowo, stosuje się kombinacje downsamplingu i upsamplingu, aby zwiększyć *receptive field* każdej warstwy konwolucyjnej. Downsampling to prosta sprawa, pooling jest rodzajem downsamplingu, który zredukuje rozmiar obrazu poprzez agregację pikseli w oknach.

8.2 Upsampling

Jest wiele sposobów, na który można robić upsampling:

- Bed of Nails - nadaj nowym komórkom wartość 0
- Nearest Neighbor - nadaj nowym komórkom wartość najbliższego sąsiada
- Bilinear Interpolation - użyj dwóch najbliższych sąsiadów w obydwu wymiarach i skonstruuj interpolację liniową
- Bicubic Interpolation - użyj trzech najbliższych sąsiadów w obydwu wymiarach i skonstruuj interpolację kubiczną
- Max-Unpooling - przy downsamplingu zapamiętujemy pozycję największych wartości w każdym oknie, a przy upsamplingu przywracamy w te pozycje wartości

Przykład 8.1

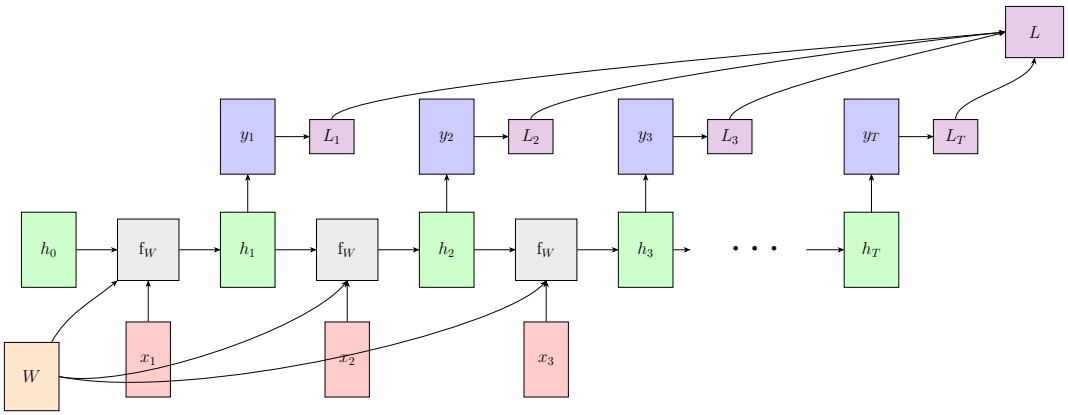
$$\begin{bmatrix} 1 & 2 & 6 & 3 \\ 3 & 5 & 2 & 1 \\ 1 & 2 & 2 & 1 \\ 7 & 3 & 4 & 8 \end{bmatrix} \Rightarrow \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} \Rightarrow \cdots \Rightarrow \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 0 & 2 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 4 \end{bmatrix}$$

9 RNN

Rekurencyjne sieci neuronowe (RNN) są sieciami neuronowymi, które przetwarzają dane sekwencyjne, gdzie każdy element sekwencji jest przetwarzany w kontekście poprzednich elementów. Kluczowym konceptem, jest tak zwany *hidden state*.

$$h_t = f_W(h_{t-1}, x_t) \quad y_t = g(h_t)$$

W zależności od problemu, możemy mieć do czynienia z tylko jednym wejściem zamiast t wejść, oraz jednym wyjściem zamiast t wyjść. To wszystko zależy od problemu, który chcemy rozwiązać.



Rysunek 1: Ilustracja architektury RNN w modelu z wieloma wejściami i wyjściami

10 NN search

W związku z trudnością projektowania sieci neuronowych, zaproponowano metody automatycznego szukania optymalnych architektury sieci. Większość tych metod opiera się o następującą architekturę:

1. Kontroler: projektuje sieci neuronowe
2. Tester: losowo wybiera architektury kontrolera, trenuje je, zwraca do kontrolera wyniki
3. Kontroler: wykonuje optymalizację na podstawie wyników testera