UWAGI od Rafala
(rafjoz@gmail.com)

Wydaje mi sie ze moze byc warto poeksperymentowac z inna parametryzacja predykcji p(y | x). W sieciach neuronowych optymalizujemy zwykle - log p(y | x), gdzie p jest modelowane jakims rozkladem prawdopodobienstwa. Jest tutaj spore pole do wyboru, w zaleznosci od rodzaju danych.

Np gdy p jest modelowane rozkladem Gaussowskim ze stala wariancja (p ~ N(y | x; std=1)), odpowiadajaca funkcja kosztu bedzie:
- log N(y | x; std=1) = - log(1/sqrt(2 * pi * std^2) - e^((y-y')^2 / 2std^2)) = 0.5 * (y-y')^2 + const == MSE(y, y'), gdzie y' to predykcja sieci neuronowej

Z powyzszego wynika ze uzywajac funkcji kosztu MSE, efektywnie zakladamy ze nasze etykiety sa dobrze modelowane rozkladem normalnym. Te zalozenie moze nie byc idealne jako, ze wiemy, iz prawdopodobienstwa katow moga miec 2 wartosci maksymalne w niektorych przypadkach – wtedy siec neuronowa zwraca najgorszy mozliwy wynik, tzn wartosc po srodku miedzy dwoma maksimami.

Mozna ten problem rozwiazac na kilka sposobow modelujac predykcje innymi rozkladami prawdopodobienstwa.

1. Najprostszy z nich, ktory byl testowany w tej pracy to podzielenie dziedziny funkcji [0;1] na wiele kubelkowow i modelowanie wynikow rozkladem dyskretnym. Rozklad ten jest duzo bardziej elastyczny i pozwala modelowac o wiele bardziej skomplikowane rozklady, ale z drugiej strony przyjmuje duzo mniej zalozen o strukturze modelowanego rozkladu. W naszym przypadku wiemy, ze nasza funkcja jest ciagla, wiec predykcje dla sasiednich wartosci powinny byc podobne. Prawdopodobnie z tego powodu moze byc trudniejsze trenowanie takiej sieci neuronowej.
(*) Mozna sprobowac z dodaniem jakichs regularyzacji do sieci ktora pomoze w treningu, np dodatkowa funkcja kosztu ktora mowi ze sasiednie predykcje musza byc podobne, tzn:
jakis_wspolczynnik * \sum (p_i - p_{i+1})^2

2. Inny sposob to uzycie mixture of gaussians jako funkcji kosztu, tzn
- log[p1 * N_1(y1 | x; std=1) + p2 * N_2(y2 | x; std=1)], p1+p2=1
Taka funkcja kosztu pozwoli nam modelowac nasz rozklad w lepszy sposob, bo bedziemy w stanie przewidziec potencjalne oba maksima rozkladu prawdopdobienstwa. W tym wypadku siec neuronowa bedzie zwracala 4 wartosci zamiast 1, tzn: [p1, p2, y1, y2] do parametryzacji obu funkcji Gaussowskich a nasza funkcja kosztu bedzie troche bardziej skomplikowana
(ale wciaz dosc prosta do optymalizacji uzywajac numerycznie stabilnej operacji logsumexp(.)):
- log p1 N1 + p2 N2 = - log[p1 / sqrt(2 pi) * e^(-(y-y1)^/2) + p2 / sqrt(2pi) * e^(-(y-y2)^2/2)] = - log[\sum_i e^[log(p_i) + log(1/sqrt(2pi)) - (y-y_i)^2/2] = - logsumexp[log(p) - (y-y')^2 / 2], gdzie p=[p1, p2], y'=[y1, y2]

Tutaj, jest przykladowa implementacja w tensorflow: https://ngbinghao.gitlab.io/posts/mixture-density-networks-basics/ dla troche ogolniejszego przypadku w ktorym dodatkowo jeszcze parametryzujemy odchylenia standardowe gaussow sieciami neuronowymi, tzn siec zwraca p=[p1, p2], y'=[y1, y2] i std=[sigma1, sigma2], co odpowiada rozkladowi:
p1 * N(mean=y1, std=sigma1 | x) + p2 * N(mean=y2, std=sigma2 | x)

Funkcja kosztu jest wtedy podobna, ale dochodzi jeszcze kilka dodatkowych skladnikow (od sqrt(2 pi sigma^2) ktore juz nie jest stala wartoscia) i ((y-y')/sigma)^2/2

Ta implementacja jest calkiem dobra, ale niestety ma jeden bug numeryczny, log(p) powinno byc wyliczone uzywajac funkcji tf.nn.log_softmax(p) zamiast tf.log(tf.nn.softmax(p)). Ta druga moze nie byc stabilna numerycznie.

W tym wariancie, podobnie jak w uzytym w raporcie, optymalizacje mozna wykonac wzgledem wartosci maksymalnej. Mozna tez sprobowac czegos bardziej skomplikowanego co wzieloby pod uwage caly rozklad zamiast tylko maksimum, ale nie wiem czy warto probowac.
Byloby to troche trudniejsze chyba i moze nie pomogac.

(*) Znalezienie maksimum wymaga tutaj sprawdzenia wartosci dla 0, 1 i w punktach y1 i y2 (przy zalozeniu ze sa w przedziale [0;1]) jako ze maksymalna wartosc moze byc w jednym z tych 4 punktow w zaleznosci od predykcji sieci.

3. Warianty 1. i 2. z parametryzowanym odchyleniem standardowym rozkladow tez mogloby byc ciekawe do sprawdzenia, ale prawdodpodobnie jest mniej wazne w tym problemie

4. Jeden problem o ktorym nie wspomnialem wczesniej jest taki ze warianty z rozkladami normalnymi nie biora pod uwage tego ze dziedzina y w naszym problemie jest zamkniety przedzial [0;1] podczas gdy dziedzina rozkladow normalnych jest cale R. Mozna to sprobowac uzyc i bylo kilka ciekawych prac na ten temat w przypadku modeli ktore generuja obrazki. Tzn intensywnosci kolorow pikseli sa modelowane 3 zmiennymi o wartosciach z przedzialu [0;255] dla kazdego z kanalow RGB.
Historycznie implementacje ktore uzywaly rozkladu normalnego do modelowania tych wartosci dawalo lepsze rezultaty niz rozklady dyskretne. Jakis czas temu ludzie zaczeli eksplorowac inne rozkladu ktore staraja sie wziac pod uwage informacje na temat tego ze dziedzina funkcji jest ograniczona. Jedna z prac jest: https://arxiv.org/abs/1701.05517 ktora uzywa mikstury zdyskretyzowanych rozkladow logistycznych, co wydaje sie dzialac bardzo dobrze na obrazkach.
Jest nawet kod zalaczony do tej pracy tutaj: https://github.com/openai/pixel-cnn . Nie jestem pewien czy warto to probowac jako ze moze byc to dosc skomplikowane, ale wciaz wydaje sie byc ciekawe i moze dac lepsze wyniki.

Inne uwagi:
* Sposob ewaluacji wydaje mi sie ze moglby byc lepszy, tzn odleglosc od maksimum nie jest idealne w sytuacji gdy rzeczywisty rozklad ma dwa maksima i w takich wypadkach odleglosc moze byc niejednoznaczna gdy mamy 2 punkty o takich samych wartosciach. Dodatkowo, wydaje mi sie ze wazniejsze/ciekawsze jest modelowanie calego rozkladu niz tylko wartosci maksymalnej, szczegolnie w sytuacjach gdzie scalar/pseduscalar (0 i 1) maja podobne wartosci a maksimum lub minimum jest gdzies po srodku. Tutaj ewaluacja by uzyla punktow 0, 0.5 lub 1 jako optymalnej i ucieka tu bardzo duzo interesujacych informacji o rzeczywistym rozkladzie.

* Inny sposob moglby po prostu ewaluowac cala entropie krzyzowa miedzy rzeczywistym rozkladem a przewidzianym. W przypadku rozkladu dyskretnego efektywnie to robimy, ale takiej samej metryki mozna uzyc we wszystkich innych rozkladach. Konkretnie, dla rozkladu normalnego, mozemy zewaluowac funkcje w 50 punktach na przedziale [0; 1], wyliczyc prawdopodobienstwa, ktore trzeba pozniej znormalizowac zeby sie sumowaly do 1 (co efektywnie ogranicza dziedzine predykcji do przedzialu [0; 1]) Mozna to pewnie zrobic porzadniej uzywajac calek jako ze znamy funkcyjna reprezentacje tych rozkladow ale moze to nie byc konieczne.

* Komentarz do sekcji 6 ("Regresja do parametrow rozkladu wag A, B, C").
Wydaje mi sie ze mogloby byc uzyteczne sprobowac polaczenie tej metody razem z poprzednimi, tzn uzywac tej regresji do A, B, C jako dodatkowej funkcji kosztu, przynajmniej na poczatku treningu. Moze to przyspieszyc uczenie i potencjalnie dostarczyc sieci neuronowej dodatkowych informacji.

* Jako ze wiemy, ze rzeczywisty rozklad uzywa funkcji sin(x) i cos(x), moze byc warto uzyc tych przeksztalcen jako nieliniowosci w samej sieci neuronowej. Konkretnie, w ostatniej warstwie ktora przeksztalcamy nieliniowoscia ReLu,
mozna dodatkowo przeksztalcic wartosci sin(x) i cos(x) i skleic je ze soba co nam da 3x wiecej wyjsc z tej konkretnej warstwy:

```
x - wyjscie poprzedniej warstwy
nowa_warstwa = tf.concat([tf.nn.relu(x), tf.sin(x), tf.cos(x)], axis=1)   # <- nowe transformacje
y = linear(nowa_warstwa, 1)
loss = tf.losses.mean_squared_error(...)
```

Ciezko powiedziec czy to pomoze, ale jest bardzo proste do sprawdzenia i moze byc uzyteczne.

Ach, jeszcze jedno - od strony praktycznej polecam przetestowac numerycznie te nowe rozklady prawdopodobienstwa jako ze latwo popelnic bledy przy definiowaniu funkcji kosztu, ale bardzo latwo sprawdzic czy wartosci tych funkcji sa poprawne uzywajac najprostszych wzorow, ktore nie musza byc bardzo stabilne numerycznie. Np, dla rozkladu normalnego lub mikstury dwoch rozkladow, bardzo latwo ewaluowac N(y'; mean=x', std=1) uzywajac Scipy lub innych bibliotek do obliczen numerycznych.