Wykorzystanie autograd do uczenia sieci neuronowej

Nie musimy już pisać logiki propagacji wstecznej!

Wszystko to może wydawać się sporym kawałkiem pracy inżynierskiej, ale wysiłek ten się opłaca. Teraz, gdy uczymy sieć neuronową, nie musimy już pisać żadnej logiki propagacji wstecznej! Jako szkolny przykład, oto sieć neuronowa z ręcznie dodaną propagacją wsteczną:

```
import numpy
np.random.seed(0)

data = np.array([[0,0],[0,1],[1,0],[1,1]])
target = np.array([[0],[1],[0],[1]])

weights_0_1 = np.random.rand(2,3)
weights_1_2 = np.random.rand(3,1)

for i in range(10):

    layer_1 = data.dot(weights_0_1)
    layer_2 = layer_1.dot(weights_1_2)

    diff = (layer_2 - target)
    sqdiff = (diff * diff)
    loss = sqdiff.sum(0)

    layer_1_grad = diff.dot(weights_1_2.transpose())
    weight_1_2_update = layer_1.transpose().dot(diff)
    weight_0_1_update = data.transpose().dot(layer_1_grad)

    weights_1_2 -= weight_1_2_update * 0.1
    weights_0_1 -= weight_0_1_update * 0.1
    print(loss[0])
```

Przewidywanie

Porównanie

Średni błąd kwadratowy (strata)

Uczenie się; tu następuje propagacja wsteczna

```
0.4520108746468352
0.33267400101121475
0.25307308516725036
0.1969566997160743
0.15559900212801492
0.12410658864910949
0.09958132129923322
0.08019781265417164
0.06473333002675746
0.05232281719234398
```

Musimy wykonywać propagację w przód w taki sposób, by `layer_1`, `layer_2` oraz `diff` istniały jako zmienne, gdyż potrzebujemy ich później. Następnie musimy propagować wstecznie każdy gradient przez odpowiednią macierz wag i wykonać aktualizację wag we właściwy sposób.

```
import numpy
np.random.seed(0)

data = Tensor(np.array([[0,0],[0,1],[1,0],[1,1]]), autograd=True)
target = Tensor(np.array([[0],[1],[0],[1]]), autograd=True)

w = list()
w.append(Tensor(np.random.rand(2,3), autograd=True))
w.append(Tensor(np.random.rand(3,1), autograd=True))

for i in range(10):

    pred = data.mm(w[0]).mm(w[1])  <-- Przewidywanie

    loss = ((pred - target)*(pred - target)).sum(0)  <-- Porównanie

    loss.backward(Tensor(np.ones_like(loss.data)))  <-- Uczenie się

    for w_ in w:
        w_.data -= w_.grad.data * 0.1
        w_.grad.data *= 0

print(loss)
```

Jednak przy naszym wspaniałym systemie autograd kod jest znacznie prostszy. Nie musimy przechowywać żadnych tymczasowych zmiennych (ponieważ ich śledzeniem zajmuje się dynamiczny graf obliczeń) i nie musimy implementować żadnej logiki propagacji wstecz (gdyż obsługuje to metoda `.backward()`). Nie tylko jest to wygodniejsze, ale mamy mniejsze możliwości popełniania głupich błędów przy pisaniu kodu propagacji wstecznej, zmniejszając ryzyko zepsucia programu!

```
[0.58128304]
[0.48988149]
[0.41375111]
[0.34489412]
[0.28210124]
[0.2254484]
[0.17538853]
[0.1324231]
[0.09682769]
[0.06849361]
```

Zanim przejdziemy dalej, chciałbym wypunktować ważną cechę stylistyczną tej nowej implementacji. Zwróćmy uwagę, że wszystkie parametry umieściłem w liście, po której mogę iterować podczas wykonywania aktualizacji wag. Jest to zapowiedź kolejnego elementu funkcjonalności. Gdy mamy już system autograd, implementacja stochastycznej metody gradientowej staje się trywialna (to po prostu ta pętla `for` na końcu ostatniego fragmentu kodu). Spróbujmy uczynić z niej jej własną klasę.