OCR_mnist

September 17, 2021

0.0.1 Klasyfikacja pojedynczych znaków

Zagadnienie jakie poruszymy w tej sekcji będzie rozpoznawanie cyfr zapisanych pisemem odręcznym.

Rozpoczniemy od załadowania niezbędnych bibliotek. Będziemy używać biblioteki "Mxnet" do trenowania i testowania sieci neuronowej, która posłuży do rozpoznawania odręcznie zapisanych cyfr pochodzących ze zbioru MNIST.

[1]:
```
import mxnet as mx
```

Przygotowanie zbioru danych Gdy mamy już załadowane niezbędne biblioteki, możemy przejść dalej do przygotowania danych. Rozpoczynamy od pobrania zbioru danych mnist z podziałem na cześć testową oraz terningową.

[2]:
```
# Fixing the random seed
mx.random.seed(42)

mnist = mx.test_utils.get_mnist()
batch_size = 100
train_data = mx.io.NDArrayIter(mnist['train_data'], mnist['train_label'],␣
 ↪batch_size, shuffle=True)
val_data = mx.io.NDArrayIter(mnist['test_data'], mnist['test_label'],␣
 ↪batch_size)
```

Wykorzystamy tradycyjną architekturę głębokich sieci neuronowych o nazwie perceptron wielowarstwowy (MLP). Rozpoczynamy od zaimportowania modułu "nn".

[3]:
```
from __future__ import print_function
from mxnet import gluon
from mxnet.gluon import nn
from mxnet import autograd as ag
```

Definicia sieć perceptron wielowarstwowy Wykorzystamy podejście wielowarstwowy perceptron do rozwiązania tego problemu. Zdefiniujemy siec MLP przy użyciu imperatywnego podejścia MXNet.

MLP składają się z kilku w pełni połączonych warstw. W pełni połączona warstwa (ang. Fully connected) lub w skrócie warstwa FC to taka, w której każdy neuron w warstwie jest połączony

z każdym neuronem w poprzedniej warstwie. Z perspektywy algebry liniowej warstwa FC stosuje transformację afiniczną do macierzy wejściowej X o rozmiarze n x m i generuje macierz wyjściową Y o rozmiarze n x k, gdzie k jest liczbą neuronów w warstwie FC. k jest również określane jako ukryty rozmiar. Wyjście Y jest obliczane zgodnie z równaniem Y = W X + b. Warstwa FC ma dwa parametry, których można się nauczyć, macierz wag W o rzomiarze m x k i wektor odchylenia b o rozmiarze m x 1 .

W MLP wyjścia większości warstw FC są podawane do funkcji aktywacji, która stosuje nieliniowość elementową. Ten krok jest krytyczny i daje sieciom neuronowym możliwość klasyfikowania danych wejściowych, które nie są liniowo rozdzielone. Typowe wybory funkcji aktywacji to sigmoid, tanh i rektyfikowana jednostka liniowa (ReLU). W tym przykładzie użyjemy funkcji aktywacji ReLU, która ma kilka pożądanych właściwości i jest zwykle uważana za wybór domyślny.

Poniższy kod deklaruje trzy w pełni połączone warstwy po 128, 64 i 10 neuronów każda. Ostatnia w pełni połączona warstwa często ma swój ukryty rozmiar równy liczbie klas wyjściowych w zestawie danych. Co więcej, te warstwy FC wykorzystują aktywację ReLU do przeprowadzania transformacji ReLU z uwzględnieniem elementów na wyjściu warstwy FC.

W tym celu użyjemy warstwy typu Sequential. To liniowy stos warstw sieci neuronowych. To nic innego jak w pełni połączone warstwy, które zostały omówione powyżej.

[4]:
```
# define network
net = nn.Sequential()
with net.name_scope():
    net.add(nn.Dense(128, activation='relu'))
    net.add(nn.Dense(64, activation='relu'))
    net.add(nn.Dense(10))
```

Inicjujemy parametry i optymalizator. Poniższy kod źródłowy inicjalizuje wszystkie parametry otrzymane z parametru dict przy użyciu inicjatora Xavier do trenowania sieci MLP, którą zdefiniowaliśmy powyżej.

Do naszego szkolenia wykorzystamy optymalizator stochastycznego gradientu (SGD). W szczególności będziemy używać mini-partii SGD. Standardowy SGD przetwarza dane o pociągach po jednym przykładzie na raz. W praktyce jest to bardzo powolne i można przyspieszyć proces, przetwarzając przykłady w małych partiach. W takim przypadku nasza wielkość partii będzie wynosić 100, co jest rozsądnym wyborem. Innym parametrem, który tutaj wybieramy, jest szybkość uczenia się, która kontroluje wielkość kroku, jaki optymalizator przyjmuje w poszukiwaniu rozwiązania. Wybierzemy współczynnik uczenia się 0,02. Ustawienia, takie jak wielkość partii i szybkość uczenia się, są zwykle nazywane hiperparametrami. Jakie wartości im nadajemy, mogą mieć ogromny wpływ na wyniki treningowe.

Użyjemy klasy Trainer, aby zastosować optymalizator SGD na zainicjowanych parametrach.

[5]:
```
gpus = mx.test_utils.list_gpus()
ctx =  [mx.gpu()] if gpus else [mx.cpu(0), mx.cpu(1)]
net.initialize(mx.init.Xavier(magnitude=2.24), ctx=ctx)
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.02})
```

Trenowanie sieci. Zazwyczaj uczenie prowadzi się do zbieżności, co oznacza, że na podstawie danych pociągu nauczyliśmy się dobrego zestawu parametrów modelu (wagi + obciążenia). Na potrzeby

tego samouczka przeprowadzimy trening przez 10 epok i zatrzymamy się. Epoka to jedno pełne przejście przez wszystkie dane pociągu.

Szkolenie będzie się odbywało poprzez następujące kroki:

1. Zdefiniuj metrykę oceny dokładności na podstawie danych uczących.
2. Zapętlaj wejścia dla każdej epoki.
3. Przekaż dane wejściowe przez sieć, aby uzyskać dane wyjściowe.
4. Straty obliczeniowe z danymi wyjściowymi i etykietą w zakresie rekordu.
5. Gradient podpory wewnątrz zasięgu rekordu.
6. Zaktualizuj metrykę i parametry oceny za pomocą opadania gradientowego.

Funkcja strat bierze pary (wyjście, etykieta) i oblicza stratę skalarną dla każdej próbki w minipartii. Skalary mierzą odległość każdego wyjścia od etykiety. Istnieje wiele predefiniowanych funkcji straty w gluon.loss. Tutaj używamy softmax_cross_entropy_loss do klasyfikacji cyfr. Obliczymy stratę i wykonamy propagację wsteczną w zakresie szkolenia zdefiniowanym przez autograd.record().

[6]:
```
%%time
epoch = 10
# Use Accuracy as the evaluation metric.
metric = mx.metric.Accuracy()
softmax_cross_entropy_loss = gluon.loss.SoftmaxCrossEntropyLoss()
for i in range(epoch):
    # Reset the train data iterator.
    train_data.reset()
    # Loop over the train data iterator.
    for batch in train_data:
        # Splits train data into multiple slices along batch_axis
        # and copy each slice into a context.
        data = gluon.utils.split_and_load(batch.data[0], ctx_list=ctx,␣
↪batch_axis=0)
        # Splits train labels into multiple slices along batch_axis
        # and copy each slice into a context.
        label = gluon.utils.split_and_load(batch.label[0], ctx_list=ctx,␣
↪batch_axis=0)
        outputs = []
        # Inside training scope
        with ag.record():
            for x, y in zip(data, label):
                z = net(x)
                # Computes softmax cross entropy loss.
                loss = softmax_cross_entropy_loss(z, y)
                # Backpropagate the error for one iteration.
                loss.backward()
                outputs.append(z)
        # Updates internal evaluation
```

```
        metric.update(label, outputs)
        # Make one step of parameter update. Trainer needs to know the
        # batch size of data to normalize the gradient by 1/batch_size.
        trainer.step(batch.data[0].shape[0])
    # Gets the evaluation result.
    name, acc = metric.get()
    # Reset evaluation result to initial state.
    metric.reset()
    print('training acc at epoch %d: %s=%f'%(i, name, acc))
```

```
training acc at epoch 0: accuracy=0.780683
training acc at epoch 1: accuracy=0.899017
training acc at epoch 2: accuracy=0.913750
training acc at epoch 3: accuracy=0.923300
training acc at epoch 4: accuracy=0.930733
training acc at epoch 5: accuracy=0.937533
training acc at epoch 6: accuracy=0.942500
training acc at epoch 7: accuracy=0.946667
training acc at epoch 8: accuracy=0.950683
training acc at epoch 9: accuracy=0.953317
CPU times: user 1min, sys: 5.25 s, total: 1min 5s
Wall time: 49.6 s
```

Predykcja Po zakończeniu powyższego szkolenia możemy ocenić wyszkolony model, uruchamiając predykcje na zestawie danych walidacji. Ponieważ zbiór danych zawiera również etykiety dla wszystkich obrazów testowych, możemy obliczyć metrykę dokładności względem danych walidacyjnych w następujący sposób.

[7]:
```
# Use Accuracy as the evaluation metric.
metric = mx.metric.Accuracy()
# Reset the validation data iterator.
val_data.reset()
# Loop over the validation data iterator.
for batch in val_data:
    # Splits validation data into multiple slices along batch_axis
    # and copy each slice into a context.
    data = gluon.utils.split_and_load(batch.data[0], ctx_list=ctx, batch_axis=0)
    # Splits validation label into multiple slices along batch_axis
    # and copy each slice into a context.
    label = gluon.utils.split_and_load(batch.label[0], ctx_list=ctx,␣
 ↪batch_axis=0)
    outputs = []
    for x in data:
        outputs.append(net(x))
    # Updates internal evaluation
    metric.update(label, outputs)
print('validation acc: %s=%f'%metric.get())
```

```
assert metric.get()[1] > 0.94
```

```
validation acc: accuracy=0.953300
```

Jeśli wszystko poszło dobrze, powinniśmy zobaczyć wartość dokładności około 0.96, co oznacza, że jesteśmy w stanie dokładnie przewidzieć cyfrę w 96% obrazów testowych.