SIECI NEURONOWE – ćwiczenie 4

W ćwiczeniu 4 odtworzymy zaimplementowaną już architekturę sieci w pełni połączonej korzystając z gotowego rozwiązania do budowania sieci neuronowych. Poniżej jest podany jest przykład uczenia w Pytorchu, inne rozwiązania są dopuszczalne (tensorflow, w jego obrębie keras który jest nawet prostszy), pod warunkiem że dopuszczają zdefiniowanie własnej architektury sieci.

Od frameworku do głębokiego uczenia oczekujemy w pierwszej kolejności:

- Implementacji operacji macierzowych/tensorowych w takim zakresie jakie będą nam potrzebne do budowy sieci
- Możliwości automatycznego wyliczania gradientów po zaimplementowanych operacjach
- Wydajnej implementacji dedykowanej GPU/TPU

Oraz typowych funkcjonalności ułatwiających budowanie sieci neuronowych z wykorzystaniem powyższych (funkcje kosztu, standardowe warstwy, optimizery itp.).

Dla torcha, konwencja jest następująca: korzystamy z obiektów **torch.tensor** do przechowywania danych, natomiast sieć zbudowana jest z modułów dziedziczących po **torch.NN.module**. Przez tensor rozumiemy tutaj n-wymiarową tablicę liczb analogicznie do numpy. Dla tensorów nadpisane są podstawowe operatory matematyczne, dostępne są operacje na tablicach podobne do dostępnych w numpy (zmiany kształtu, transpozycje, agregacje takiej jak średnia i suma etc.). W obiekcie tensora obok właściwej wartości może być zapisany również gradient. Gradient, jeżeli istnieje, zawsze będzie tensorem o tym samym wymiarze co właściwe dane tensora. Jest inicjalizowany przy pierwszym przejściu propagacji wstecz przez dany tensor.

Model powinien dziedziczyć po **torch.NN.module** i definiować **__init__()** – operacje wykonywane przy tworzeniu instancji klasy oraz **forward()** – funkcję która na podstawie danych wejściowych zwraca wyjście modelu.

Oficjalny tutorial odnośnie tego, jak uczyć model, opisuje pętlę uczącą w ten sposób: https://pytorch.org/tutorials/beginner/introyt/trainingyt.html

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9) #1

def train_one_epoch(epoch_index, tb_writer):
    running_loss = 0.
    last_loss = 0.

    # Here, we use enumerate(training_loader) instead of
    # iter(training_loader) so that we can track the batch
    # index and do some intra-epoch reporting
    for i, data in enumerate(training_loader):
        # Every data instance is an input + label pair
        inputs, labels = data

        # Zero your gradients for every batch!
        optimizer.zero_grad() #2
```

```
        # Make predictions for this batch
        outputs = model(inputs) #3

        # Compute the loss and its gradients
        loss = loss_fn(outputs, labels) #4
        loss.backward() #5

        # Adjust learning weights
        optimizer.step() #6

    return loss
```

Jak i dlaczego to działa?

1. Obiekt optimizer służy do optymalizowania danego zbioru parametrów. Parametry są podklasą tensora specjalnie uwzględnianymi przez moduły.

   `parameters()` dowolnego obiektu `torch.NN.module` zwraca parametry przynależące do tejże klasy oraz parametry każdej zmiennej przynależącej, która sama jest podklasą `torch.NN.module`. Uwaga: funkcja nie zadziała na zmiennych zagnieżdżonych jeśli jakikolwiek poziom nie jest podklasą `torch.NN.module`! Na przykład kod:

   ```
   __init__(self):
        super(TinyModel, self).__init__()
        self.layer_list = [torch.nn.Linear(10,10) for i in range(123)]
   ```

   Stworzy listę 123 warstw liniowych 10x10, ale jako że jedyną zmienną przynależącą do klasy jest sama lista, model nie będzie poprawnie uwzględniał tych 123 warstw i ich macierzy wag w `parameters()`.

2. `zero_grad()` jest potrzebne, ponieważ moduł automatycznego różniczkowania akumuluje, a nie nadpisuje gradienty przy wielokrotnym wywołaniu.

3. Wywołanie instancji modelu jak funkcji (pythonowe __call__) jest równoważne forward()

4. Funkcje kosztu są dostępne jako parametryzowalne klasy w torch.nn. To znaczy, że najpierw instancjonujemy je jako obiekt, a potem wywołujemy loss_fn() na odpowiednich tensorach odpowiadających wejściom i pożądanym wyjściom – należy zwrócić uwagę na oczekiwany typ tensora w dokumentacji i pamiętać o domyślnych formatach! (W szczególności, domyślny float torcha NIE zgadza się z domyślnym w numpy – 32 vs 64bit!)

5. backward() to całe przejście wstecz w algorytmie propagacji wstecznej. Jak udało się to tak uprościć? Po pierwsze, każda operacja na tensorach dostępna w torchu, analogicznie do sposobu implementacji sugerowanego w poprzednim ćwiczeniu, ma zarówno funkcję forward(), jak i backward(). Po drugie, operacje w trakcie przechodzenia w przód są dołączane do *grafu obliczeniowego*

zawierającego informacje o tym skąd wejście do danej operacji. Istnienie grafu obliczeniowego pozwala przeiterować po wcześniej wywołanych operacjach wstecz i wyliczyć gradienty na każdym poziomie, oraz zakumulować je w obiektach parametrów.

6. `step()` dokonuje zmiany parametrów zgodnie z wartością gradientu i przyjętą regułą uczenia. Dla SGD jest to klasyczne -learning_rate*gradient

W ćwiczeniu należy zbudować sieć o tych samych parametrach co w zadaniu poprzednim, i ocenić jej działanie na tych samych danych, tym razem sprawdzając wpływ:

- wybranego optimizera (SGD i dwa inne)
- rozmiaru batcha
- wartości współczynnika uczenia dla różnych optimizerów

Ćwiczenie oceniane jest w skali 0-10 pkt.