Zadanie 02

Krzysztof Dziechciarz

Wyciskanie soku

1. Ładowanie zbioru CIFAR-10 w TensorFlow

```
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
```

W powyższy sposób załadowałem zbiór testowy pod obiektem `trainset`, pobierając i zapisując go w folderze `./data`

2. Kilka obrazów

1. Przykładowe obrazy ze zbioru CIFAR-10

3, 4. Stworzenie sieci neuronowej analogicznej do LeNet-5 i wypisanie jej właściwości

Korzystając z frameworka `pytorch` zbudowałem sieć analogiczną jak na rysunku w instrukcji. Jedyną różnicą była liczba kanałów wejściowych dla pierwszej warstwy. Wynosi ona 3, bo klasyfikujemy obrazy kolorowe.

```
net = LeNet5CIFAR10()

print(net)

params = 0
for parameter in net.parameters():
    subsum=1
    for el in parameter.size():
        subsum*=el
    params+=subsum

print(params)
```

```
LeNet5CIFAR10(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
62006
```

Jak policzyłem wyżej, sieć ma 62006 parametrów.

5. Inicjalizacja sieci

Framework PyTorch domyślnie inicjalizuje sieć odpowiednimi wartościami w zależności od użytej funkcji aktywacji. Dla ReLu jest to inicjalizacja He, a biasy są ustawiane na 0. Wynika z tego, że w moim przypadku nie jest wymagane pisanie żadnego

kodu odpowiedzialnego za inicjalizację.

6-10. Zapętlenie w treningu

Zaimplementowałem pętlę uczącą używając entropii krzyżowej jako funkcji kosztu. Ze zbioru treningowego wydzieliłem przez losowanie jego część na potrzeby walidacji. Trening przeprowadzam wykonując kolejne epoki w batchach 64-elementowych. Dane zostały wymieszane, a jako optymalizator wybrałem Adam.

Działanie pętli treningowej:

- przełączenie modelu w tryb treningu
- ładowanie danych z kolejnego batcha
- zerowanie gradientów
- przepuszczenie nowych danych przez sieć
- obliczenie funkcji kosztu
- obliczenie gradientu funkcji kosztu
- modyfikacja wag modelu przu użyciu optymalizatora tak, aby zminimalizować funkcję kosztu

Pętla po każdym batchu wypisuje aktualne statystyki uczenia:

```
Epoch 1/10, Training Loss: 2.1599, Validation Loss: 2.1094, Elapsed time: 0:01:03.735386
Epoch 2/10, Training Loss: 2.0876, Validation Loss: 2.0618, Elapsed time: 0:01:36.780064
Epoch 3/10, Training Loss: 2.0553, Validation Loss: 2.0585, Elapsed time: 0:02:08.809854
Epoch 4/10, Training Loss: 2.0425, Validation Loss: 2.0649, Elapsed time: 0:02:39.885934
Epoch 5/10, Training Loss: 2.0284, Validation Loss: 2.0167, Elapsed time: 0:03:13.197674
Epoch 6/10, Training Loss: 2.0238, Validation Loss: 2.0410, Elapsed time: 0:03:44.163056
Epoch 7/10, Training Loss: 2.0125, Validation Loss: 2.0136, Elapsed time: 0:04:17.162842
Epoch 8/10, Training Loss: 2.0056, Validation Loss: 2.0191, Elapsed time: 0:04:50.307064
Epoch 9/10, Training Loss: 2.0051, Validation Loss: 2.0240, Elapsed time: 0:05:24.624843
Epoch 10/10, Training Loss: 1.9940, Validation Loss: 1.9915, Elapsed time: 0:05:58.220890
```

A po zakończeniu działania obliczam accuracy na całym zbiorze testowym:

```
Test Accuracy: 49.16%
```

Nie jest to wynik zadowalający i mam nadzieje, że w kolejnych krokach uda się go znacząco ulepszyć.

11-14. Hiperparameryzacja sieci

Zamknąłem dotychczasowy kod wewnąrz funkcji:

```python
def train_and_eval(learning_rate, num_layers, layer_sizes, batch_size, dropout_rate, weight_decay):
```

Parametrami wejściowymi były:

- learning rate
- liczba warstw w pełni połączonych
- lista wielkości tych warstw
- batch size
- dropout rate
- weight decay (regularyzacja L2)

Następnie skonfigurowałem funkcję biblioteki Optuna by przeszukiwać przestrzeń hiperparametrów:

```python
def objective(trial):
    learning_rate = trial.suggest_float('learning_rate', 1e-5, 1e-2, log=True)
    dropout_rate = trial.suggest_float('dropout_rate', 0.0, 0.5)
    weight_decay = trial.suggest_float('weight_decay', 1e-5, 1e-2, log=True)
    num_layers = int(trial.suggest_float("num_layers", 2.0, 10.0, step=1.0, log=False))
```

```
    batch_size = trial.suggest_categorical('batch_size', [16, 32, 64, 128, 256])

    layer_sizes = []
    layer_sizes.append(int(trial.suggest_float("layer-1-size", 128, 1024, step=1.0, log=False)))
    for i in range(1, num_layers-1):
      name = "layer-{}-size".format(i+1)
      layer_sizes.append(int(trial.suggest_float(name, 64, 1024, step=1.0, log=False)))

    layer_sizes.append(10)

    print(f"Hyperparameters of trial number {trial.number}")
    print(f"  Learning Rate: {learning_rate}")
    print(f"  Dropout Rate: {dropout_rate}")
    print(f"  Weight Decay: {weight_decay}")
    print(f"  Number of Layers: {num_layers}")
    print(f"  Batch Size: {batch_size}")
    print(f"  Layer Sizes: {layer_sizes}")

    accuracy = train_and_eval(
        learning_rate,
        num_layers,
        layer_sizes,
        batch_size,
        dropout_rate,
        weight_decay
    )
    print()
    print(f"  Accuracy: {accuracy}")
    print(f"  Elapsed time: {datetime.datetime.now() - trial.datetime_start}")
    print()
    print()
    print()

    return accuracy
```

Gdy mamy już funkcję, której parametry chcemy optymalizować, podajemy Optunie zakres wartości, w którym będzie szukała optymalnych wartości hiperparametrów.

```
optuna.logging.get_logger("optuna").addHandler(logging.StreamHandler(sys.stdout))
study_name = "mro-kadet-04"
storage_name = "sqlite:///{}.db".format(study_name)
study = optuna.create_study(study_name=study_name, storage=storage_name, direction="maximize")

study.optimize(objective, n_trials=1000)

print("Najlepsze hiperparametry: ", study.best_params)
print("Najlepsza dokładność: ", study.best_value)
```

Następnie skonfigurowałem `study` optuny tak, aby wyniki kolejnych prób były zapisywane w bazie danych i można było kontynuować obliczenia z dowolnego punktu optymalizacj.

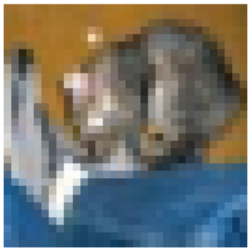
Udało mi się wykonać 240 rund optymalizacji, co trwało ok 17h i dało najlepszy wynik `accuracy=0.6923` w 84. próbie przy parametrach:

```
Hyperparameters of trial number 84
  Learning Rate: 0.0003261243021880896
  Dropout Rate: 0.06525664504539422
  Weight Decay: 1.3502174401630838e-05
  Number of Layers: 2
  Batch Size: 32
  Layer Sizes: [943, 10]
```

W końcowych kilkudziesięciu próbach wartości hiperparametrów potrafiły różnić się od siebie znacząco, jednak wartości accuracy cały czas znajdowały się w przedziale od 60% do 70%.
Optuna pomogła ulepszyć wynik sieci, było to ulepszenie istotne, ale nie tak wysokie jak się spodziewałem, biorąc pod uwagę czas jaki był potrzebny na wykonanie tak dużej liczby prób.

15. Kilka obrazków

True: cat | Predicted: dog



True: ship | Predicted: automobile

True: ship | Predicted: ship

True: airplane | Predicted: airplane

True: frog | Predicted: deer