

Dwuwarstwowy model rozpoznawania kwiatów irysa oraz drzewo decyzyjne do tego problemu.

Komentarze w kodzie z przyzwyczajenia będą w j. ang. 😊

Model będzie się uczyć na danych z:

<https://www.kaggle.com/datasets/uciml/iris>

Zacznę od spojrzenia na dane. Nasze dane prezentują się w ten sposób:

	sepal.length	sepal.width	petal.length	petal.width	variety
001	5.1	3.5	1.4	.2	Setosa
002	4.9	3	1.4	.2	Setosa
003	4.7	3.2	1.3	.2	Setosa
004	4.6	3.1	1.5	.2	Setosa
005	5	3.6	1.4	.2	Setosa
006	5.4	3.9	1.7	.4	Setosa
007	4.6	3.4	1.4	.3	Setosa

Mamy 4 wartości float, które są opisem naszego kwiata, następnie mamy kolumnę variety, która mówi nam z jakim rodzajem kwiata mamy do czynienia. Mamy 3 rodzaje kwiatów, są to: Setosa, Virginica i Versicolor.

Musimy więc nasze kolory zmapować na liczby, aby nasz model mógł je przyjąć.

```
# Data reading process
iris_df = pd.read_csv("iris.csv")

# print(iris_df["variety"].unique())

# All data must be in number form, so we do mapping with variety
iris_df["variety"] = iris_df["variety"].map({'Setosa':0,'Versicolor':1,'Virginica':2})

# print(iris_df.head())

# Preparing data and target
X = iris_df.drop(["variety"],axis=1).values
y = iris_df["variety"].values
```

Wykonałem od razu szybki podział na dane oraz klasy kwiatów które będziemy klasyfikować – X i y odpowiednio.

Problem nie jest zbyt skomplikowany i model w łatwy sposób jest w stanie się go nauczyć. Mamy tutaj około 150 rekordów kwiatów, więc dobrym pomysłem walidacji naszego modelu byłoby podzielenie setu na dane treningowe oraz testowe do walidacji. Podzieliłem set 70/30, metodą train_test_split(). Stratify jest tylko po to aby w każdym z setów było więcej niż 0 każdego rodzaju kwiata.

```
# Splitting it for train and test 70/30
train_X, test_X, train_y, test_y = train_test_split(X, y, test_size=0.3, stratify=y)
```

Teraz zajmę się wczytaniem danych do Loaderów, którymi łatwo będę mógł iterować po rekordach, przy okazji wdrażając podział na ‘batche’.

Na początek zamienię sobie dane na tensory i wczytam do CPU lub GPU.

```
# Mapping it to tensors, adding to CPU or GPU
train_X = torch.FloatTensor(train_X).to(device)
test_X = torch.FloatTensor(test_X).to(device)
train_y = torch.LongTensor(train_y).to(device)
test_y = torch.LongTensor(test_y).to(device)
```

Teraz stworzę sobie zmodyfikowaną klasę Dataset, na potrzeby naszych danych w której umieszcę nasze dane. Z tej pozycji banalnym będzie zamienienie danych na loadery.

```
# Preparing a custom dataset for this
class IrisDataset(Dataset):
    def __init__(self, features, labels):
        self.features = features
        self.labels = labels

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, idx):
        return self.features[idx], self.labels[idx]

train_dataset = IrisDataset(train_X, train_y)
test_dataset = IrisDataset(test_X, test_y)
```

```
train_dataset = IrisDataset(train_X, train_y)
test_dataset = IrisDataset(test_X, test_y)
```

Metody `__len__` oraz `__getitem__` są to metody wymagane przy tworzeniu customowego Datasetu.

W międzyczasie zdefiniuje sobie dwa hiperparametry których będę używał:

```
# Hyperparams
LR = 0.001
BATCH_SIZE = 32
EPOCHS = 100
```

Teraz mogę bezpośrednio stworzyć dataloadery na danych:

```
# Creating a data loader for easy iteration
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False)
```

Dane przygotowane, więc mogę się teraz zabrać za tworzenie modelu.

Model będzie mało skomplikowany – warstwa wejścia, warstwa ukryta oraz wyjścia, z funkcją ReLU jako funkcją aktywacji. Wszystkie warstwy będą liniowe:

```
# Creating our model, with 1 hidden layer and ReLU
class Classifier(nn.Module):
    def __init__(self):
        super(Classifier, self).__init__()
        self.linear1 = nn.Linear(4, 128)
        self.linear2 = nn.Linear(128, 64)
        self.linear3 = nn.Linear(64, 3)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.linear1(x))
        x = self.relu(self.linear2(x))
        x = self.linear3(x)
        return x
```

Definiuje sobie funkcje straty oraz optymalizator.

W tym przypadku użyję CrossEntropyLoss, który jest powszechnie stosowany w problemach klasyfikacji. Jest to funkcja logarytmiczna, która mierzy jak daleko jest przewidywanie od prawdy.

Co do optymalizatora, użyje do tego problemu Adam (Adaptive Moment Estimation), który automatycznie dostosowuje współczynniki uczenia dla każdego parametru w modelu na podstawie gradientów oraz ich średnich kwadratów.

```
# Defining optimizer and loss fn
model = Classifier()
model.to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(params=model.parameters(), lr=LR)
```

Możemy przejść do właściwego treningu modelu.

Do treningu użyje funkcji train:

```
# Model training
def train():
    model.train()
    loss_values = []

    for i in range(1,EPOCHS+1):
        loss = 0.0
        for _, data in enumerate(train_loader):
            inputs, label = data

            optimizer.zero_grad()
            pred = model(inputs)

            loss = criterion(pred, label)
            loss_values.append(loss)
            loss.backward()
            optimizer.step()

        print(f"Epoch: {i} Train loss: {loss}")
```

Dzięki data loaderom mogę po prostu iterować po danych, wydzielając input i label z każdego rekordu. Liczymy dla każdego rekordu wartość straty oraz używam optymalizatora aby dostosować wagi po treningu przez propagację wsteczną.

Po każdej epoce wyświetlę jaką wartość ma strata, aby sprawdzić czy model się uczy

Teraz mogę wykorzystać ewaluację, aby zobaczyć jak sprawuje się model na danych testowych.

Obliczam sobie przy tym średnią wartość straty oraz celność modelu.

```
# Model evaluation
def eval():
    model.eval()

    total_loss = 0
    correct_predictions = 0
    total_samples = 0
    with torch.no_grad():
        for _, data in enumerate(test_loader):
            inputs, label = data
            pred = model(inputs)
            loss = criterion(pred, label)
            total_loss += loss.item()
            _, predicted = torch.max(pred.data, 1)
            total_samples += label.size(0)
            correct_predictions += (predicted == label).sum().item()

    avg_loss = total_loss / len(test_loader)
    accuracy = correct_predictions / total_samples

    print(f"Test Loss: {avg_loss} Accuracy: {accuracy}")
```

Model po wytrenowaniu, mogę bez problemu zapisać w pliku .pth, który zapisuje lokalnie.

Oto rezultat treningu oraz testu na 100 epokach:

```
Data loader from CSV file, mapping values...
All values has been mapped, prepping data...
Data correctly prepared
Starting training...
Epoch: 5 Train loss: 0.6917831897735596
Epoch: 10 Train loss: 0.26707494258880615
Epoch: 15 Train loss: 0.17825405299663544
Epoch: 20 Train loss: 0.31011858582496643
Epoch: 25 Train loss: 0.15946778655052185
Epoch: 30 Train loss: 0.05671803280711174
Epoch: 35 Train loss: 0.0399264395236969
Epoch: 40 Train loss: 0.18553362786769867
Epoch: 45 Train loss: 0.06726092100143433
Epoch: 50 Train loss: 0.27487391233444214
Epoch: 55 Train loss: 0.04167359322309494
Epoch: 60 Train loss: 0.08920516818761826
Epoch: 65 Train loss: 0.05496389418840408
Epoch: 70 Train loss: 0.025265924632549286
Epoch: 75 Train loss: 0.04578787460923195
Epoch: 80 Train loss: 0.09360204637050629
Epoch: 85 Train loss: 0.013265151530504227
Epoch: 90 Train loss: 0.008732527494430542
Epoch: 95 Train loss: 0.01603798009455204
Epoch: 100 Train loss: 0.08202419430017471
Model saved
Starting evaluation process...
Test Loss: 0.051797155290842056 Accuracy: 1.0
```

Możemy od razu zauważyć, że 100 epok jest to stanowczo za dużo, model jest w stanie nauczyć się problemu w znacznie mniejszym czasie, potem nie robi za wiele postępu. Dataset nie jest zbyt duży więc nie mamy za wiele do zmiany, spróbuje zmniejszyć epoki oraz pobawić się trochę hiperparametrami.

Learning rate wydaje się być poprawny, przy mniejszym model nie uczy się zejść poniżej 0.15 loss, tak aby nie zajęło mu to 400 epok, przy większym, widać mocne wahania w działaniu.

Postanowiłem zmienić Batch size oraz ilość epok, prezentuję się to następująco:

```
Starting training...
Epoch: 5 Train loss: 0.5363325476646423
Epoch: 10 Train loss: 0.42100346088409424
Epoch: 15 Train loss: 0.2277073860168457
Epoch: 20 Train loss: 0.2145053893327713
Epoch: 25 Train loss: 0.16120436787605286
Epoch: 30 Train loss: 0.10478377342224121
Epoch: 35 Train loss: 0.09657667577266693
Epoch: 40 Train loss: 0.06752897799015045
Model saved
Starting evaluation process...
Test Loss: 0.08460729569196701 Accuracy: 0.9777777777777777
```

Model był w stanie nauczyć się w bardzo krótkim czasie na podobny poziom, przez zmianę batch size. Zmniejszenie batch size wydłuża czas działania, jednak program uczy się w mniej epok. Jednak główną zaletą batchy, jest to jak program jest w stanie generalizować to co się nauczył. Tutaj jest to mało widoczne, jednak przy bardziej złożonych problemach, możemy w ten sposób unikać zjawisk przeuczenia.

Tak prezentuję się prosty model i jego proces uczenia na równie prostym problemie jakim jest klasyfikacja kwiatów irysa.

Sprawdźmy teraz jak z takim problemem poradzi sobie drzewo decyzyjne zaprojektowane dla tego problemu.

Drzewo decyzyjne na podstawie danych klasyfikacji kwiatów irysa.

Wykorzystam do tego te same dane, tak samo je podzielię na train/test w proporcjach 70/30.

Tak samo jak wcześniej, wczytałem dane i przy pomocy sklearn podzieliłem je. Następnym krokiem było już tylko stworzenie drzewa i wrzucenie do niego danych.

```
PATH = 'iris.csv'

# Loading
iris_df = pd.read_csv("iris.csv")

# All data must be in number form, so we do mapping with variety
iris_df["variety"] = iris_df["variety"].map({'Setosa':0,'Versicolor':1,'Virginica':2})

# Preparing data and target
X = iris_df.drop(["variety"],axis=1).values
y = iris_df["variety"].values

# print(X)

# Splitting it for train and test 70/30
train_X, test_X, train_y, test_y = train_test_split(X, y, test_size=0.3, stratify=y)
```

Po eksperymentowałem nieco z głębokością drzewa i w naszym przypadku, głębokość 3 będzie zupełnie wystarczająca.

```
for depth in range(1,8):
    decision_tree = tree.DecisionTreeClassifier(max_depth = depth)
    decision_tree.fit(train_X, train_y)

    y_pred = decision_tree.predict(test_X)
    acc_decision_tree = round(decision_tree.score(test_X, test_y) * 100, 2)
    print(f'Depth: {depth}, tree acc: {acc_decision_tree}')
```

```
Depth: 1, tree acc: 66.67
Depth: 2, tree acc: 88.89
Depth: 3, tree acc: 95.56
Depth: 4, tree acc: 95.56
Depth: 5, tree acc: 95.56
Depth: 6, tree acc: 95.56
Depth: 7, tree acc: 95.56
```

Ostateczny wynik działania drzewa na danych testowych, prezentuję się następująco:

Depth: 3, tree acc: 97.78

Jest to wynik bardzo zbliżony do wyniku naszego modelu. W naszym przypadku problem jest bardzo prosty, więc nie będziemy widzieć różnic w działaniu obu modelki.

Same drzewo prezentuję się następująco:

