

## Zadania na 3.0 – sprawozdanie

Wszystkie zadania będą wykonywane przeze mnie przy użyciu Pythona i jego bibliotek. Komentarze w kodzie będę wykonywał w języku angielskim.

1. Stworzenie drzewa decyzyjnego na datasetcie titanic (dostępny pod linkiem: <https://www.kaggle.com/competitions/titanic>)

Do wykonania tego zadania użyję biblioteki Scikit-Learn, do wczytania i manipulacji danymi będę używał Pandas.

Otwieram zestaw danych, do treningu i do testu, dane są w postaci plików CSV

```
PATH_TRAIN = 'train.csv'
PATH_TEST = 'test.csv'

train = pd.read_csv(PATH_TRAIN)
test = pd.read_csv(PATH_TEST)

# print(train.head(5))
```

Interesuje mnie zestaw do treningu, z którego muszę wydzielić dane oraz target, którym będzie kolumna Survived.

Zanim jednak to zrobię, warto byłoby uzupełnić brakujące dane wiekowe niektórych pasażerów. Wiek w tym przypadku pełni dosyć ważną rolę jeśli rozważamy przetrwanie pasażerów. Pustych danych jest całkiem sporo, co mogę łatwo sprawdzić sumując je:

```
print('NaN: ', sum(train['Age'].isna()), ' / ', len(train['Age']))
```

```
NaN: 177 / 891
```

Puste dane postanowiłem uzupełnić, grupując pasażerów kolumnami ['Survived', 'Pclass', 'Sex'] i wyciągając średnią wiekową z tych grup.

Uzupełniam dane przy użyciu .fillna():

```
l_col = ['Survived', 'Pclass', 'Sex']

train['Age'] = train['Age'].fillna(train.groupby(l_col)['Age'].transform('mean'))
```

Przed wrzuceniem danych do klasyfikatora, trzeba je jeszcze nieco wyczyścić oraz zamienić wartości słowne na liczbowe. Dodaje nową kolumnę FamilySize oraz IsAlone, rezygnując z niepotrzebnych.

```
# Mapping Cabins
train['Has_Cabin'] = train["Cabin"].apply(lambda x: 0 if type(x) == float else 1)
test['Has_Cabin'] = test["Cabin"].apply(lambda x: 0 if type(x) == float else 1)

# Create new feature FamilySize as a combination of SibSp and Parch
for dataset in full_data:
    dataset['FamilySize'] = dataset['SibSp'] + dataset['Parch'] + 1

# Create new feature IsAlone from FamilySize
for dataset in full_data:
    dataset['IsAlone'] = 0
    dataset.loc[dataset['FamilySize'] == 1, 'IsAlone'] = 1

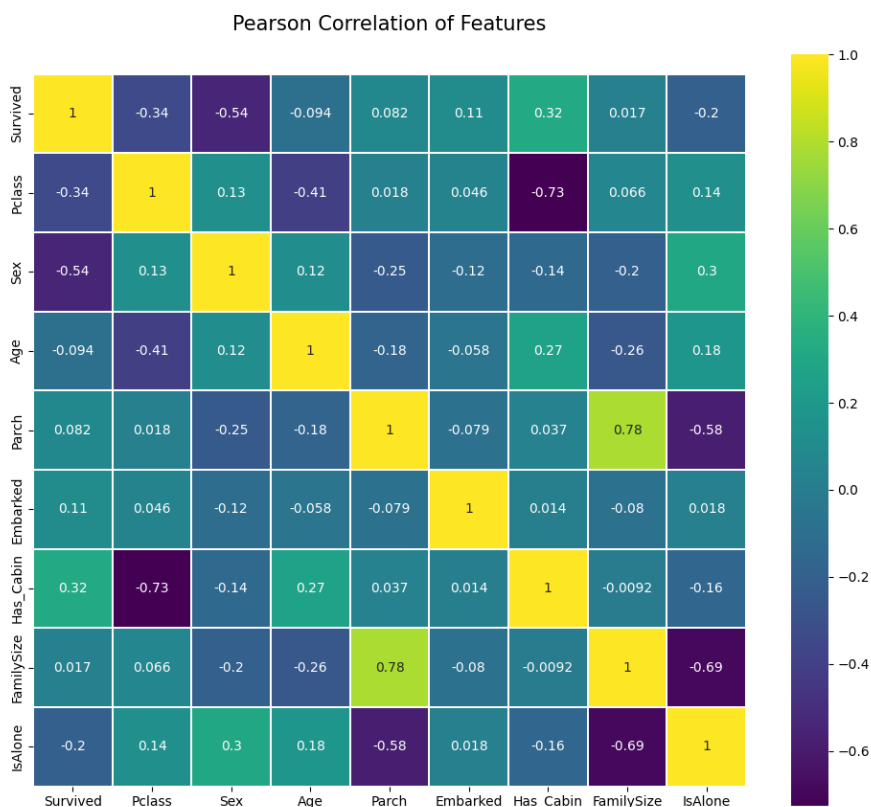
# Remove all NULLS in the Embarked column
for dataset in full_data:
    dataset['Embarked'] = dataset['Embarked'].fillna('S')

# Mapping Embarked
dataset['Embarked'] = dataset['Embarked'].map( {'S': 0, 'C': 1, 'Q': 2} ).astype(int)

# Mapping Sex
dataset['Sex'] = dataset['Sex'].map( {'female': 0, 'male': 1} ).astype(int)

drop_elements = ['PassengerId', 'Name', 'Ticket', 'Cabin', 'SibSp', 'Fare']
train = train.drop(drop_elements, axis = 1)
test = test.drop(drop_elements, axis = 1)
print(train.head(4))
```

Przy użyciu biblioteki Seaborn, można w fajny sposób zwizualizować korelacje między danymi:



Chciałbym teraz sprawdzić, jak głębokie drzewo będzie najbardziej optymalne w naszym przypadku. Do tego, użyję sobie techniki Cross Validation, która będzie dzielić nasz train set na części i obliczać celność dla każdej z nich.

Możliwe głębokości ustawiłem na od 1 do ilości kategorii w datasetcie, ilość podziałów na 10:

```
cv = KFold(n_splits=10) # Desired number of Cross Validation folds
accuracies = list()
max_attributes = len(list(test))
depth_range = range(1, max_attributes + 1)

# Testing max_depths from 1 to max attributes
for depth in depth_range:
    fold_accuracy = []
    tree_model = tree.DecisionTreeClassifier(max_depth = depth)
    print("Current max depth: ", depth, "\n")
    for train_fold, valid_fold in cv.split(train):
        f_train = train.loc[train_fold] # Extract train data with cv indices
        f_valid = train.loc[valid_fold] # Extract valid data with cv indices

        model = tree_model.fit(X = f_train.drop(['Survived'], axis=1),
                               y = f_train['Survived']) # We fit the model with the fold train data
        valid_acc = model.score(X = f_valid.drop(['Survived'], axis=1),
                                y = f_valid['Survived']) # We calculate accuracy with the fold validation data
        fold_accuracy.append(valid_acc)

    avg = sum(fold_accuracy)/len(fold_accuracy)
    accuracies.append(avg)
    print("Accuracy per fold: ", fold_accuracy, "\n")
    print("Average accuracy: ", avg)
    print("\n")

df = pd.DataFrame({"Max Depth": depth_range, "Average Accuracy": accuracies})
df = df[["Max Depth", "Average Accuracy"]]
print(df.to_string(index=False))
```

Max Depth	Average Accuracy
1	0.786729
2	0.766654
3	0.795768
4	0.818265
5	0.833983
6	0.826092
7	0.814919
8	0.820474

Widzę, że najlepiej poszło na 5, także wykorzystam to jako max głębokość swojego drzewa.

Robię sobie podział na dane i target:

```
y_train = train['Survived']
X_train = train.drop(['Survived'], axis=1).values
X_test = test.values

print(X_train, y_train)
print(X_test)
```

Tworzę drzewo, wrzucam do niego dane, robię test na testowych danych.

Wytrenowany model zapisuje sobie, dodatkowo zapisuje sobie png z moim drzewem w celu wizualizacji:

```
decision_tree = tree.DecisionTreeClassifier(max_depth = 5)
decision_tree.fit(X_train, y_train)

y_pred = decision_tree.predict(X_test)
✓ submission = pd.DataFrame({
    "PassengerId": PassengerId,
    "Survived": y_pred
})

# submission.to_csv('predict.csv', index=False)

✓ dot_tree = tree.export_graphviz(decision_tree,
    out_file=None,
    max_depth = 5,
    impurity = True,
    feature_names = list(train.drop(['Survived'], axis=1)),
    class_names = ['Died', 'Survived'],
    rounded = True,
    filled= True )

graph = graphviz.Source(dot_tree)
graph.format = "png"
graph.render("file_name")

acc_decision_tree = round(decision_tree.score(X_train, y_train) * 100, 2)
print(acc_decision_tree)
```

Do stworzenia obrazu użyłem biblioteki Graphviz

Obraz drzewa zapisany jest w pliku tree\_image (za szeroki by tu wkleić).

Na koniec liczę sobie celność modelu, która wyniosła 86.08.

## 2. Prosta sieć dwuwarstwowa ucząca się XORA

XOR – czyli alternatywa rozłączna, poniższa tabela prezentuje działanie tej bramki logicznej:

Input A	Input B	XOR Output
0	0	0
0	1	1
1	0	1
1	1	0

Model, który stworze będzie modelem dwuwarstwowym, wejście + 1 ukryta + wyjście. Wejście będzie posiadać 2 neurony – na 1 i 0, warstwa ukryta może być w tym przypadku prosta, na 3 lub 4 neurony. Wyjście na 1 neuron jest wystarczające, będzie to float od 0 do 1, który zostanie zaokrąglony do najbliższej liczby.

Do implementacji potrzebna mi będzie biblioteka PyTorch.

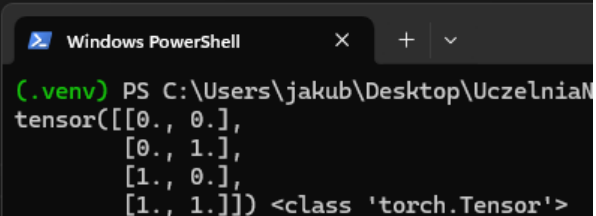
Z racji, że XOR jest łatwym problemem, w tym przypadku nie będzie konieczne robienie dużego datasetu:

```
# XOR data and targets
X = [[0, 0], [0, 1], [1, 0], [1, 1]]
y = [[0], [1], [1], [0]]

# Parsing data into torch tensors (tensor is just a word for multi dimensional vector)

inputs = torch.tensor(X, dtype=torch.float32)
targets = torch.tensor(y, dtype=torch.float32)

print(inputs, type(inputs))
```



```
(.venv) PS C:\Users\jakub\Desktop\UczelniaN
tensor([[0., 0.],
        [0., 1.],
        [1., 0.],
        [1., 1.]]) <class 'torch.FloatTensor'>
```

Tworzę sobie prosty model sieci dwuwarstwowej:

```
class XOR(nn.Module):  
  
    def __init__(self):  
        super(XOR, self).__init__()  
        self.fc1 = nn.Linear(2, 3)  
        self.fc2 = nn.Linear(3, 1)  
  
    def forward(self, x):  
  
        x = F.sigmoid(self.fc1(x))  
        x = self.fc2(x)  
        return x
```

Teraz gdy mamy nasz model, zostało zinicjalizowanie go. Następnie musimy też wybrać funkcję straty i optymalizatora.

Jako f-cje straty użyję MSE Loss ( Mean Squared error Loss), która jest stosowana do zadań regresji, jako iż XOR możemy potraktować jako regresję binarną.

Co do optymalizatora użyję SGD (Stochastic Gradient Descent), który jest prostym i powszechnie używanym optymalizatorem.

Przy bardziej skomplikowanych problemach, warto poeksperymentować z doбором obu funkcji, jednak tutaj z racji prostoty problemu, nie będzie to potrzebne.

Przy problemach i przy tworzeniu sieci, często tworzy się blok w kodzie, dedykowany stworzeniu hiperparametrów, czyli parametrów modelu, które będą decydować o jego nauce. Są to parametry takie jak: ilość epok, learning rate(jak szybko model ma się uczyć), wielkość warstw modelu, ilość neuronów w warstwie, wielkość batchy czy foldów w przypadku Cross Validation itd...

Odpowiedni dobór tych parametrów stanowi wyzwanie przy tworzeniu bardziej zaawansowanych modeli, stosuje się różne techniki aby jak najlepiej dopasować je w celu osiągnięcia najwyższej celności modelu.

W kodzie będzie to wyglądać następująco:

```
# Model init
xor_net = XOR()

# Hyperparams block
lr = 0.01
epoch_count = 8000

# Loss and optimization functions
criterion = nn.MSELoss()
optimizer = optim.SGD(xor_net.parameters(), lr=lr)
```

Mamy wszystko czego potrzebujemy, teraz wytrenujemy nasz model przy użyciu prostej pętli:

```
print("Training...")

# We set the model to training mode
xor_net.train()

for epoch in range(1, epoch_count+1):

    for input, target in zip(inputs, targets):
        optimizer.zero_grad()          # Zeroing the gradients
        out = xor_net(input)            # Getting the output from the network

        loss = criterion(out, target)    # Calculating the loss value from current output
        loss.backward()                 # Back propagation
        optimizer.step()                 # Update the weights

    if (epoch+1) % 1000 == 0:
        print(f'Epoch [{epoch+1}/{epoch_count}], Loss: {loss.item():.4f}')
```

Nasz model dla każdego zestawu danych, wylicza wynik. Następnie liczona jest strata, przy pomocy MSE oraz model wylicza gradienty dla parametrów. Przy pomocy gradientów, model jest w stanie zobaczyć jak zmieni się wartość funkcji przy zmianie parametru, przez co jest w stanie się uczyć. Na końcu następuje aktualizacja wag.

Na koniec, szybka walidacja naszego modelu przy podstawowych danych:

```
# Setting into evaluation mode for validation
xor_net.eval()

print("")
print("Final results:")
for input, target in zip(inputs, targets):
    output = xor_net(input)
    print("Input:[{}{}] Target:[{}] Predicted:[{}] Error:[{}]"
          .format(
            int(input.data.numpy()[0]),
            int(input.data.numpy()[1]),
            int(target.data.numpy()[0]),
            round(float(output.data.numpy()[0]), 4),
            round(float(abs(target.data.numpy()[0] - output.data.numpy()[0])), 4)
          ))
```

Tak oto prezentuję się wynik działania modelu - uczenie + walidacja:

```
Training....
Epoch [1000/8000], Loss: 0.2777
Epoch [2000/8000], Loss: 0.2723
Epoch [3000/8000], Loss: 0.2681
Epoch [4000/8000], Loss: 0.2624
Epoch [5000/8000], Loss: 0.2456
Epoch [6000/8000], Loss: 0.1594
Epoch [7000/8000], Loss: 0.0111
Epoch [8000/8000], Loss: 0.0001

Final results:
Input:[0,0] Target:[0] Predicted:[0.0048] Error:[0.0048]
Input:[0,1] Target:[1] Predicted:[0.9952] Error:[0.0048]
Input:[1,0] Target:[1] Predicted:[0.9917] Error:[0.0083]
Input:[1,1] Target:[0] Predicted:[0.0076] Error:[0.0076]
```

Dany problem był dosyć trywialny, dlatego szybko był on w stanie się nauczyć poprawnego rozpoznawania oraz rozwiązywania XORa.

Widzimy że model konsekwentnie zmniejszał wartość błędu po przejściu przez kolejne epoki. Można by trenować model również na więcej epok, ale myślę że mija się to z celem, przez to że model będzie w stanie osiągnąć 0 wartość błędu.



### 3. Sieć konwolucyjna ucząca się MNIST

Sieć konwolucyjna jest to rodzaj sieci neuronowej, która skutecznie analizuje dane przestrzenne, takie jak obrazy. Działa poprzez przekształcanie wejściowego obrazu za pomocą serii warstw konwolucyjnych.

Wykorzystując PyTorch stworzę sieć neuronową, która będzie posiadać jedną warstwę konwolucyjną oraz warstwy liniowe. Dla prostego problemu jakim jest MNIST, będzie to wystarczające

Zaczynam od tworzenia modelu:

```
# CONV => RELU => POOL => FC => RELU => FC => SOFTMAX
class ConvNet(nn.Module):

    def __init__(self, channels, classes) -> None:
        super(ConvNet, self).__init__()

        self.relu = nn.ReLU()

        # First set of layers
        self.conv = nn.Conv2d(in_channels=channels, out_channels=20, kernel_size=(5,5))
        self.maxpool = nn.MaxPool2d(kernel_size=(2,2), stride=(2,2))

        # Why 2880? 28x28, kernel is 5x5 so the out after kernel will be 24x24,
        # then maxpool (2x2) so the image is 2 times smaller,
        # hence 64x144 for each out_channel (64 for batch size)(144*20=2880)

        # Linear layer
        self.fc1 = nn.Linear(in_features=2880, out_features=500)

        # Softmax classifier
        self.fc2 = nn.Linear(in_features=500, out_features=classes)
        self.soft = nn.LogSoftmax(dim=1)

    def forward(self, x):
        x = self.conv(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = self.relu(x)

        x = self.fc2(x)
        out = self.soft(x)

        return out
```

Definiuje warstwy. Ważne, aby warstwa konwolucyjna miała zdefiniowaną ilość kanałów (1 – szarość, 3 – RGB) wielkość jądra oraz pooling i jego wymiary.

Warstwa ta będzie przekształcać nasz obraz na kilka mniejszych obrazów (dokładnie 20). Pooling jest to proces łączenia podobnych sąsiednich pixeli (w tym przypadku 2x2) w jeden.

Jako funkcję aktywacji wybrałem ReLu, jest prosta ( $\max(0, x)$ ) oraz szybka w trenowaniu.

Warstwy liniowe dopasowuje aby odpowiednie macierze mogły się przemnożyć.

Na końcu wykonuje softmax, czyli skalowanie wyników sieci na prawdopodobieństwa sumujące się do 1.

Ważną warstwą jest też Flatten Layer, warstwa przekształca naszą wielowymiarową warstwę w sieci w jednowymiarowy wektor. Robimy to po to, aby dopasować dane wejściowe w pełni połączonej warstwy do klasyfikacji.

Od razu też mogę zdefiniować hiperparametry:

```
# Hyperparams
LR = 0.001
BATCH_SIZE = 64
EPOCHS = 5

TRAIN_SPLIT = 0.75
VAL_SPLIT = 1 - TRAIN_SPLIT
MODEL_PATH = 'model.pth'
```

Teraz zajmę się danymi. MNIST jest bardzo popularnym datasetem, więc część bibliotek posiada wbudowaną możliwość jego pobrania:

```
# Loading the dataset
print("Loading MNIST dataset..")
train_data = MNIST(root='data', train=True, download=True, transform=ToTensor())
test_data = MNIST(root='data', train=False, download=True, transform=ToTensor())
```

Definiuję ścieżkę zapisu, części datasetu i przekształcam je od razu na Tensory, czyli po prostu wielowymiarowe wektory.

Postanowiłem, że train set, podzielę 3/1 na train oraz validation set, którym będę od razu obliczał celność modelu przy trenowaniu.

```
# Splitting into training + validation + testing
new_train_count = int(train_count * TRAIN_SPLIT)
new_val_count = int(train_count * VAL_SPLIT)

(train_data, val_data) = torch.utils.data.random_split(train_data, [new_train_count, new_val_count],
                                                         generator=torch.Generator().manual_seed(52))
print('Data after splitting:')
print(f'Train data count: {len(train_data)}, Validation data count: {len(val_data)}, Test data count: {test_count}')
```

Następnym istotnym krokiem w tworzeniu sieci w PyTorchu, który znacznie ułatwia pracę, jest stworzenie data loaderów dla każdego zbioru danych. Pozwoli to na prostą iterację po danych podczas treningu:

```
# Creating data loaders for each dataset, shuffle is enabled for train set for better generalization
train_data_loader = DataLoader(train_data, batch_size=BATCH_SIZE, shuffle=True)
val_data_loader = DataLoader(val_data, batch_size=BATCH_SIZE)
test_data_loader = DataLoader(test_data, batch_size=BATCH_SIZE)

# Calculating steps for later model evaluation
train_steps = len(train_data_loader.dataset) // BATCH_SIZE
val_steps = len(val_data_loader.dataset) // BATCH_SIZE
```

Definiuję model oraz potrzebną funkcję straty i optymalizator, tworzę też słownik na zapis danych przy treningu. W tym przypadku posłużę się adaptacyjnym optymalizatorem.

```
# Model init channels=1 for grayscale, channels=3 for RGB, moving the model to CPU or GPU
model = ConvNet(channels=1, classes=len(train_data.dataset.classes)).to(device)

# Optimizer + Loss function
optimizer = torch.optim.Adam(model.parameters(), lr=LR)
loss_fn = nn.NLLLoss()

# Preparing a dictionary for learning history
H = {
    "train_loss": [],
    "train_acc": [],
    "val_loss": [],
    "val_acc": []
}
```

Można zacząć trening. Podzieliłem trening i test na dwie osobne funkcje w kodzie.

Trenuję model i zapisuję jego wyniki, jak również poprawne odpowiedzi

```
109 def train():
110     for epoch in range(EPOCHS):
111         model.train() # Training mode for model
112
113         # Variables for calculating accuracy later
114         total_train_loss = 0
115         total_val_loss = 0
116
117         correct_train = 0
118         correct_val = 0
119
120         for (x, y) in train_data_loader:
121             optimizer.zero_grad()
122
123             (x, y) = (x.to(device), y.to(device)) # Value and class should be moved to CPU or GPU
124
125             prediction = model(x)
126             loss = loss_fn(prediction, y)
127             loss.backward()
128             optimizer.step()
129
130             total_train_loss += loss
131             correct_train += (prediction.argmax(1) == y).type(torch.float).sum().item()
```

Przy każdej epoce, ewaluuje również jego sprawność walidacją:

```
135 with torch.no_grad():
136
137     model.eval()
138
139     for (x, y) in val_data_loader:
140
141         (x, y) = (x.to(device), y.to(device))
142
143         prediction = model(x)
144
145         total_val_loss += loss_fn(prediction, y)
146         correct_val += (prediction.argmax(1) == y).type(torch.float).sum().item()
147
148
149     avg_train_loss = total_train_loss / train_steps
150     avg_val_loss = total_val_loss / val_steps
151
152     correct_train = correct_train / len(train_data_loader.dataset)
153     correct_val = correct_val / len(val_data_loader.dataset)
154
155     H["train_loss"].append(avg_train_loss.cpu().detach().numpy())
156     H["train_acc"].append(correct_train)
157     H["val_loss"].append(avg_val_loss.cpu().detach().numpy())
158     H["val_acc"].append(correct_val)
159
160     print("[INFO] EPOCH: {}/{}".format(epoch + 1, EPOCHS))
161     print("Train loss: {:.6f}, Train accuracy: {:.4f}".format(avg_train_loss, correct_train))
162     print("Val loss: {:.6f}, Val accuracy: {:.4f}\n".format(avg_val_loss, correct_val))
163
164
165     plot()
166     torch.save(model, MODEL_PATH)
```

Całość treningu i walidacji prezentuję się następująco:

```
Loading MNIST dataset..
Train data count: 60000, Test data count: 10000
Data after splitting:
Train data count: 45000, Validation data count: 15000, Test data count: 10000
[INFO] EPOCH: 1/5
Train loss: 0.178341, Train accuracy: 0.9471
Val loss: 0.068672, Val accuracy: 0.9800

[INFO] EPOCH: 2/5
Train loss: 0.056496, Train accuracy: 0.9829
Val loss: 0.058821, Val accuracy: 0.9804

[INFO] EPOCH: 3/5
Train loss: 0.034117, Train accuracy: 0.9894
Val loss: 0.045922, Val accuracy: 0.9853

[INFO] EPOCH: 4/5
Train loss: 0.023664, Train accuracy: 0.9924
Val loss: 0.053978, Val accuracy: 0.9843

[INFO] EPOCH: 5/5
Train loss: 0.017138, Train accuracy: 0.9947
Val loss: 0.047001, Val accuracy: 0.9861
```

5 Epok przy takim datasetcie jest wystarczające w zupełności.

Zapisuje sobie mój model lokalnie oraz funkcję rysuje sobie wykres na podstawie danych ze słownika.



Na koniec, funkcją test, sprawdzam jak sobie poradzi na danych testowych po jego nauczaniu.

```
def test():  
  
    print("Found saved model, evaluating on test set...")  
  
    torch.load(MODEL_PATH)  
  
    with torch.no_grad():  
        model.eval()  
        preds = []  
  
        for (x, _) in test_data_loader:  
            x = x.to(device)  
  
            prediction = model(x)  
            preds.extend(prediction.argmax(axis=1).cpu().numpy())  
  
    print(classification_report(test_data.targets.cpu().numpy(),  
                               np.array(preds), target_names=test_data.classes))
```

Korzystając z sklearn drukuje sobie również raport, jak sobie poradził model:

```
Found saved model, evaluating on test set...
```

	precision	recall	f1-score	support
0 - zero	0.99	1.00	0.99	980
1 - one	1.00	1.00	1.00	1135
2 - two	0.98	0.99	0.99	1032
3 - three	1.00	0.98	0.99	1010
4 - four	0.99	0.99	0.99	982
5 - five	0.99	1.00	0.99	892
6 - six	0.99	0.99	0.99	958
7 - seven	0.98	0.99	0.98	1028
8 - eight	0.99	0.97	0.98	974
9 - nine	0.98	0.99	0.98	1009
accuracy			0.99	10000
macro avg	0.99	0.99	0.99	10000
weighted avg	0.99	0.99	0.99	10000

Poszło mu bardzo dobrze, jest w stanie prawie ze 100% pewnością zgadywać cyfry z obrazka.

MNIST jest dosyć prostym problemem, sieci konwolucyjne stanowią bardzo dobre narzędzie do tego typu zadań. Jest to obecnie standard przy rozpoznawaniu obrazów.